Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Уральский федеральный университет имени первого Президента России Б.Н. Ельцина»

Институт радиоэлектроники и информационных технологий – РТФ

Лабораторная работа №5
Форматирование и линтинг проекта. Сборка образа проекта

Студент группы РИМ – 150950: _____ Вальнева А.Д.

Екатеринбург 2025

Цель работы

Познакомится со способами поддержки качества кода и сборки образа приложения.

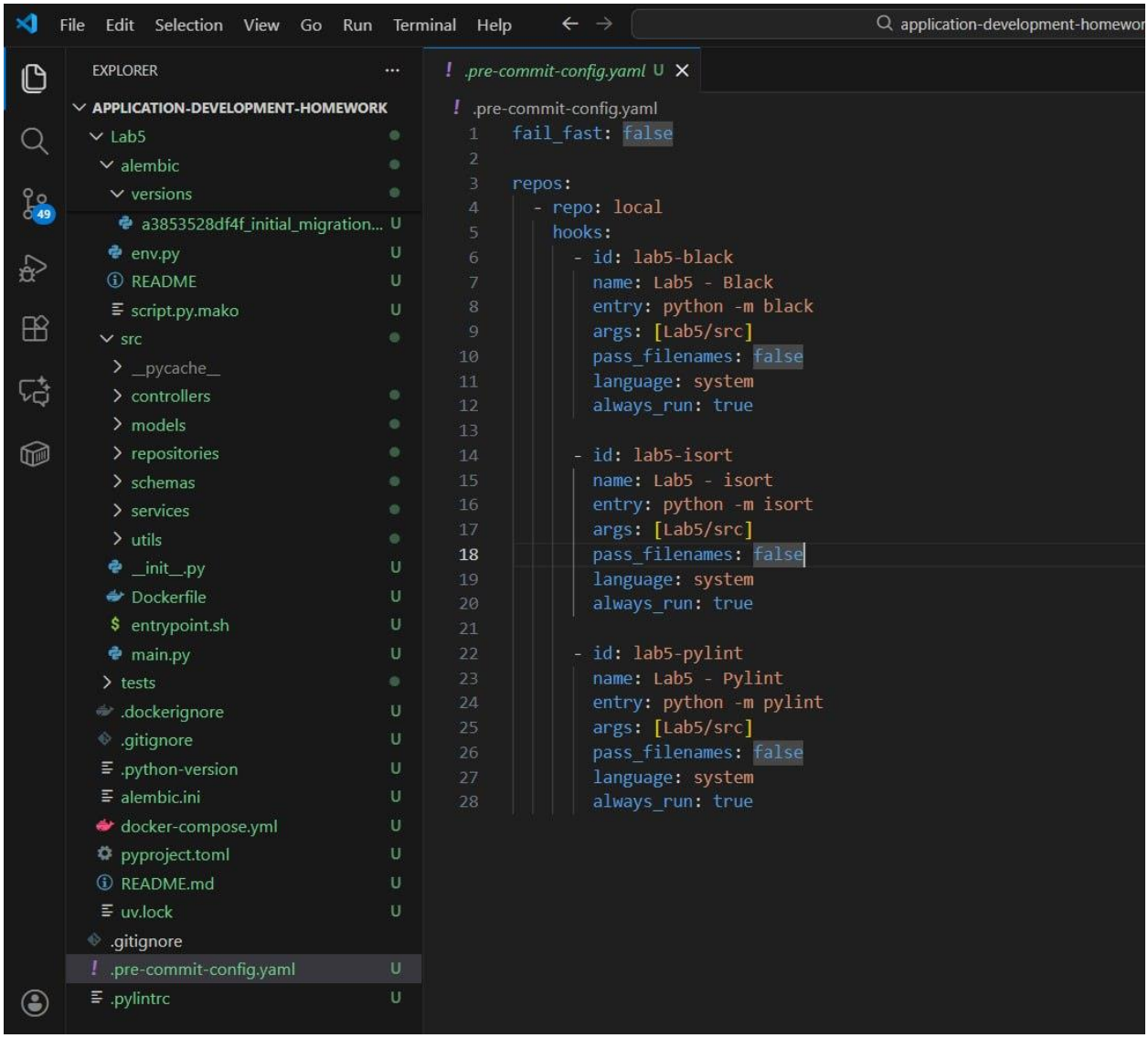
Задачи

1. **Внедрить инструменты обеспечения качества кода — подключить линтеры и форматтеры**, чтобы поддерживать единый стиль и автоматически обнаруживать типичные ошибки в ходе разработки.
2. **Выполнить pre-commit для всех файлов**, устранить предупреждения.
3. Контейнеризовать приложение (**Dockerfile уже был реализован в проекте до этого, здесь добавили entrypoint.sh**)
4. Собрать и запустить контейнеры (docker compose up --build) и проверить работу приложения на порту 8000

Ход выполнения

Часть 1: Применение линтеров и форматеров при локальной разработке

Файл:*.pre-commit-config.yaml*



Файл *.pre-commit-config.yaml* содержит конфигурацию для автоматической проверки кода перед каждым коммитом.

Описание конфигурации:

Параметр fail_fast: false позволяет выполнить все проверки, даже если одна из них завершится с ошибкой, что помогает увидеть все проблемы в коде за один запуск.

В секции repos определены локальные хуки (repo: local), которые запускают три инструмента:

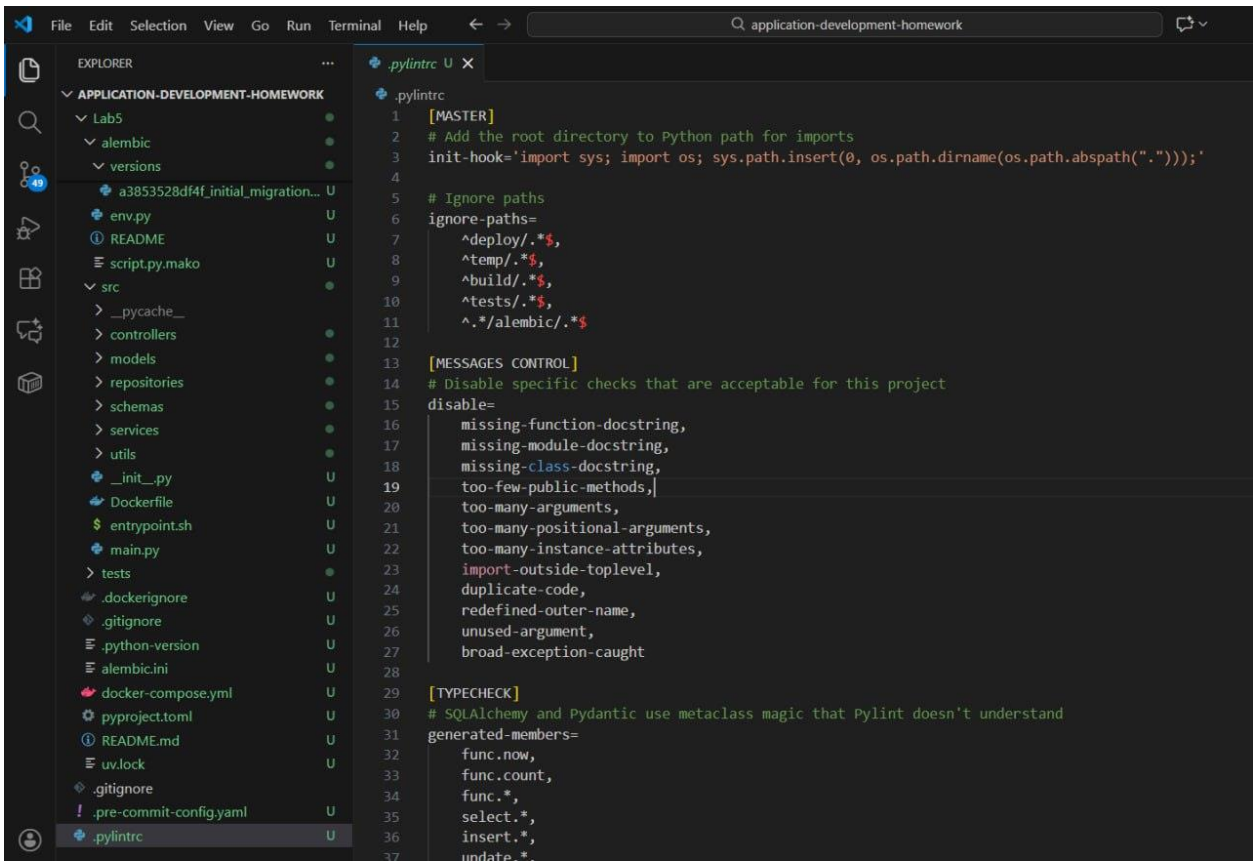
1. Black (lab5-black):
   - Автоматический форматер кода Python
   - Команда запуска: python -m black
   - [Lab5/src] - указывает на директорию с исходным кодом проекта.
   - Параметр pass_filenames: false: означает, что Black будет обрабатывать всю указанную директорию, а не отдельные файлы.
   - Параметр always_run: true: обеспечивает запуск проверки при каждом коммите.

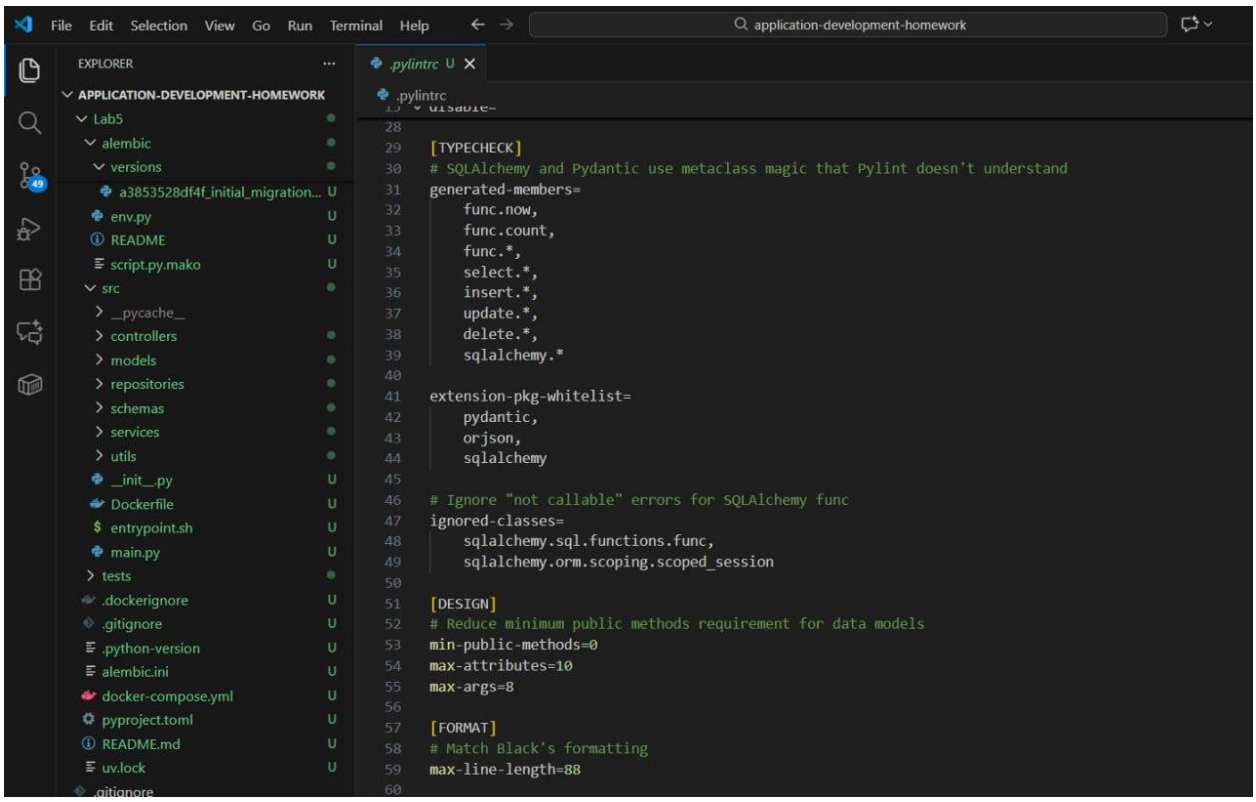
2. isort (lab5-isort):
   - Автоматическая сортировка импортов в Python файлах
   - Команда запуска: python -m isort
   - [Lab5/src] - директория с исходным кодом
   - Параметры аналогичны Black для обеспечения единообразной обработки

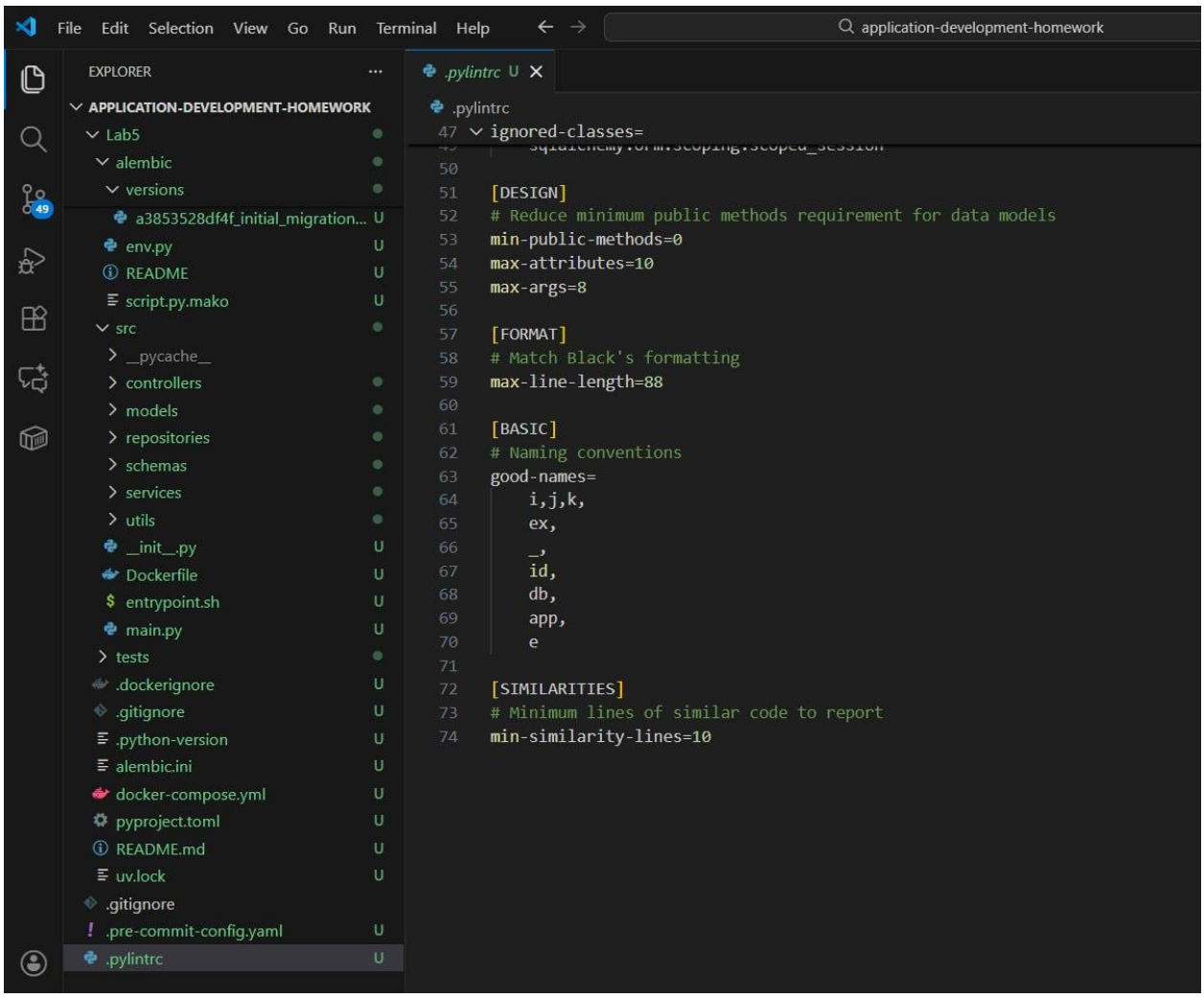
3. Pylint (lab5-pylint):
   - Статический анализатор кода для выявления ошибок и проблем с качеством кода
   - Команда запуска: python -m pylint
   - [Lab5/src] - директория для анализа

Все хуки используют language: system, что означает использование системного интерпретатора Python с установленными в нем пакетами.

Файл: *.pylintrc*
(для настройки работы линтера)


File Edit Selection View Go Run Terminal Help
application-development-homework
EXPLORER
APPLICATION-DEVELOPMENT-HOMEWORK
Lab5
alembic
versions
a3853528df4f_initial_migration... U
env.py U
README U
script.py.mako U
src
__pycache__
controllers
models
repositories
schemas
services
utils
__init__.py U
Dockerfile U
entrypoint.sh U
main.py U
tests
.dockerignore U
.gitignore U
.python-version U
alembic.ini U
docker-compose.yml U
pyproject.toml U
README.md U
uv.lock U
.gitignore
.pre-commit-config.yaml U
.pylintrc U
.pylintrc U
.pylintrc
[MASTER]
Add the root directory to Python path for imports
init-hook='import sys; import os; sys.path.insert(0, os.path.dirname(os.path.abspath(".")));'
Ignore paths
ignore-paths=
^deploy/.*$,
^temp/.*$,
^build/.*$,
^tests/.*$,
^.*/alembic/.*$
[MESSAGES CONTROL]
Disable specific checks that are acceptable for this project
disable=
missing-function-docstring,
missing-module-docstring,
missing-class-docstring,
too-few-public-methods,
too-many-arguments,
too-many-positional-arguments,
too-many-instance-attributes,
import-outside-toplevel,
duplicate-code,
redefined-outer-name,
unused-argument,
broad-exception-caught
[TYPECHECK]
SQLAlchemy and Pydantic use metaclass magic that Pylint doesn't understand
generated-members=
func.now,
func.count,
func.*,
select.*,
insert.*,


File Edit Selection View Go Run Terminal Help
application-development-homework
EXPLORER
APPLICATION-DEVELOPMENT-HOMEWORK
.pylintrc U
.pylintrc
[TYPECHECK]
SQLAlchemy and Pydantic use metaclass magic that Pylint doesn't understand
generated-members=
func.now,
func.count,
func.*,
select.*,
insert.*,
update.*,
delete.*,
sqlalchemy.*
extension-pkg-whitelist=
pydantic,
orjson,
sqlalchemy
Ignore "not callable" errors for SQLAlchemy func
ignored-classes=
sqlalchemy.sql.functions.func,
sqlalchemy.orm.scoping.scoped_session
[DESIGN]
Reduce minimum public methods requirement for data models
min-public-methods=0
max-attributes=10
max-args=8
[FORMAT]
Match Black's formatting
max-line-length=88



Описание конфигурации:

Секция [MASTER]:

- init-hook: код выполняется перед запуском Pylin, добавляет корневую директорию проекта в sys.path, что позволяет Pylint корректно разрешать импорты модулей проекта.

- ignore-paths: определяет директории, которые не должны проверяться:
  - ^deploy/.*$ - файлы развертывания
  - ^temp/.*$ - временные файлы
  - ^build/.*$ - файлы сборки
  - ^tests/.*$ - тестовые файлы
  - ^.*/alembic/.*$ - файлы миграций Alembic

Секция [MESSAGES CONTROL]:

В disable - указание проверок, которые были отключены для проекта:

- missing-function-docstring, missing-module-docstring, missing-class-docstring - отключены требования к документированию;
- too-few-public-methods - отключена проверка минимального количества публичных методов
- too-many-arguments, too-many-positional-arguments, too-many-instance-attributes - отключены ограничения на количество аргументов и атрибутов; *(например в проекте используются модели данных (User, Product, Order), которые могут иметь много атрибутов, но мало методов.)*
- import-outside-toplevel, duplicate-code, redefined-outer-name, unused-argument, broad-exception-caught - отключены специфические проверки, которые в контексте проекта создают ложные срабатывания;

Секция [TYPECHECK]:

- **generated-members:** список динамически генерируемых членов SQLAlchemy, которые Pylint должен игнорировать при проверке атрибутов (например, func.now, func.count, операции select, insert, update, delete)
- **extension-pkg-whitelist:** список пакетов-расширений C, для которых разрешена загрузка: pydantic, orjson, sqlalchemy
- **ignored-classes:** классы SQLAlchemy, для которых игнорируются ошибки типа "not callable"

Секция [DESIGN]:

- min-public-methods=0 - минимальное количество публичных методов снижено до 0 (для моделей данных)
- max-attributes=10 - максимум 10 атрибутов в классе
- max-args=8 - максимум 8 аргументов в функции

Секция [FORMAT]:

- max-line-length=88 - установлена максимальная длина строки 88 символов (соответствует стандарту Black)

Секция [BASIC]:

- **good-names:** список коротких имен переменных, которые считаются допустимыми: i, j, k (счетчики), ex (исключения), _ (неиспользуемые переменные), id, db, app, e

Секция [SIMILARITIES]:

- min-similarity-lines=10 - минимальное количество строк похожего кода для создания предупреждения о дублировании

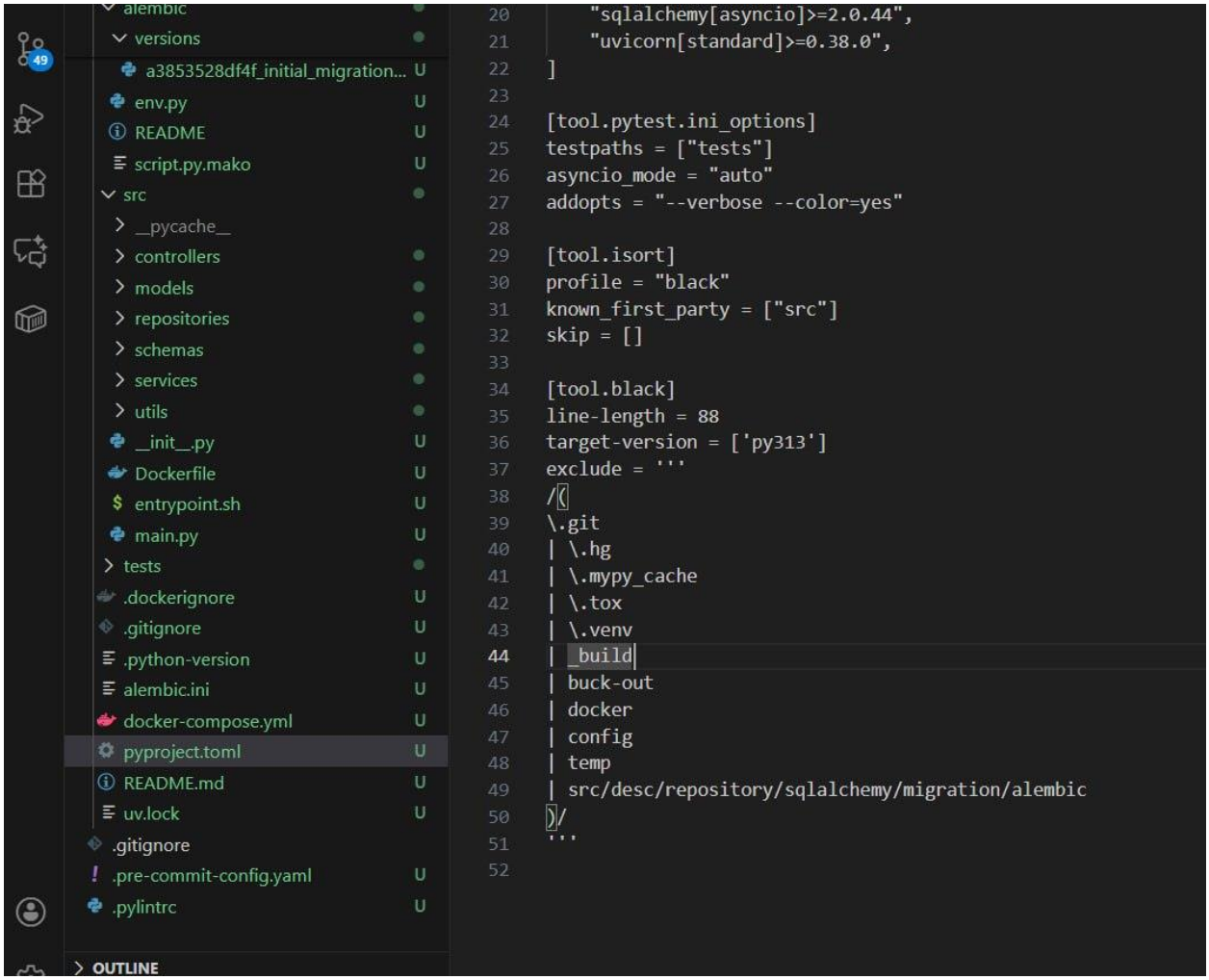
Установка pre-commit хуков **(**pre-commit install активирует автоматические проверки перед каждым коммитом)

Первоначальная проверка pre-commit run --all-files для всех файлов проекта

Теперь при каждой попытке создания коммита автоматически запускаются Black, isort и Pylint, что обеспечивает поддержание единого стиля кода и раннее выявление потенциальных проблем.

Созданный файл конфигурации .pylintrc адаптирует работу Pylint под специфику веб-приложения на FastAPI/Litestar с использованием SQLAlchemy и Pydantic.

Файл: *pyproject.toml*



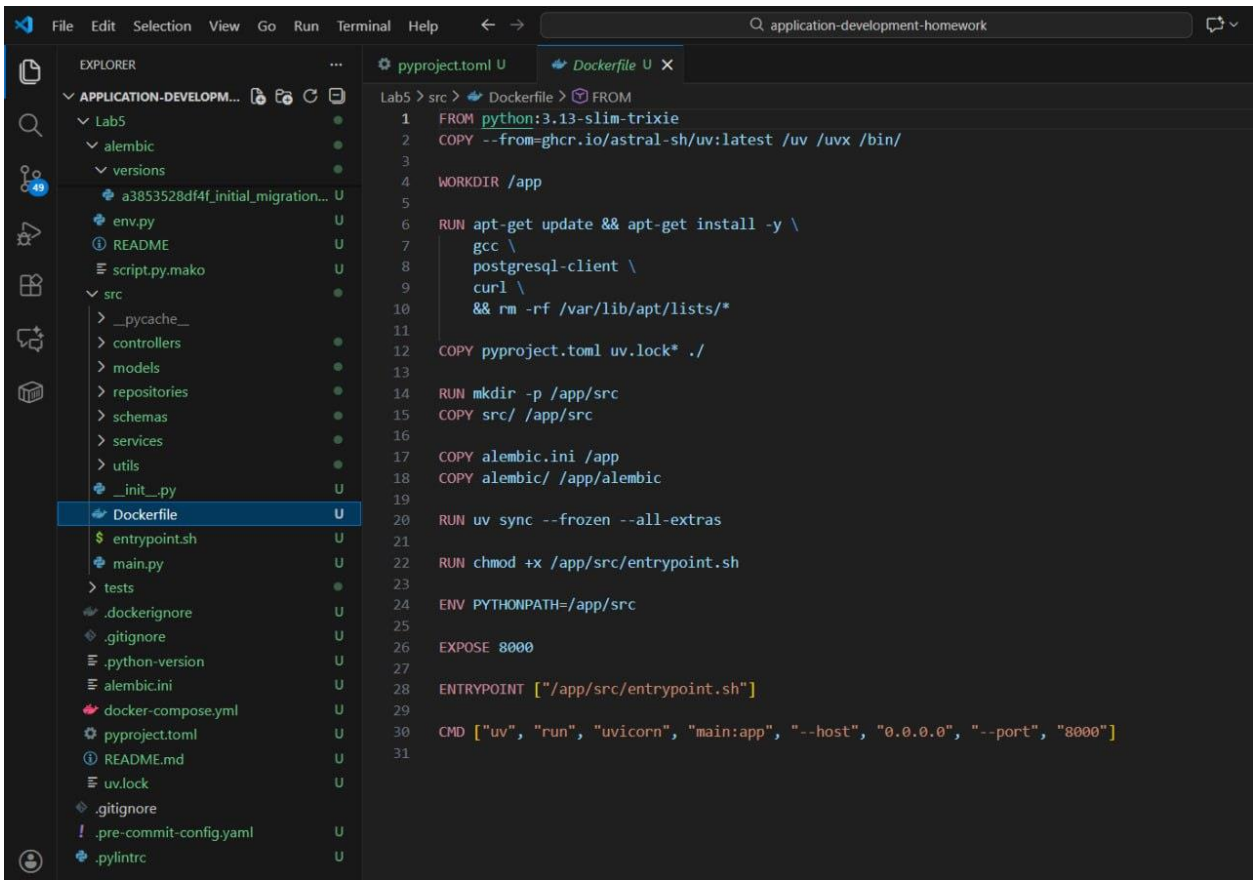
Добавлены конфигурации

- isort: совместимость с black, локальные пакеты — src.
- black: длина строки 88, целевая Py 3.13, исключения для служебных папок.

Часть 2: Сборка образа проекта с использованием Docker

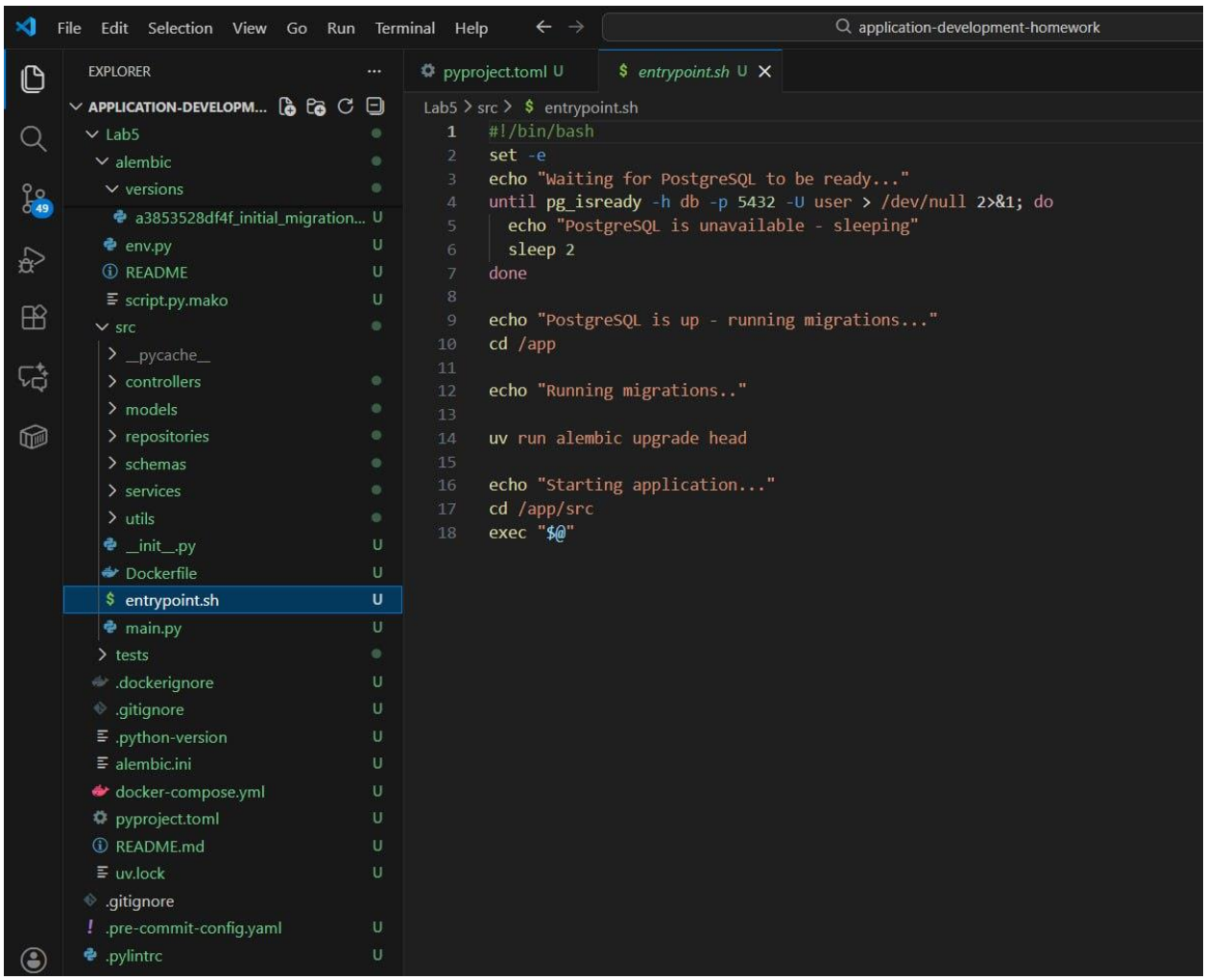
Docker используется для запуска проекта (был добавлен в 3-й лабе)

- **Настройка точки входа: указывается скрипт entrypoint.sh для запуска приложения**



Файл*: entrypoint.sh*

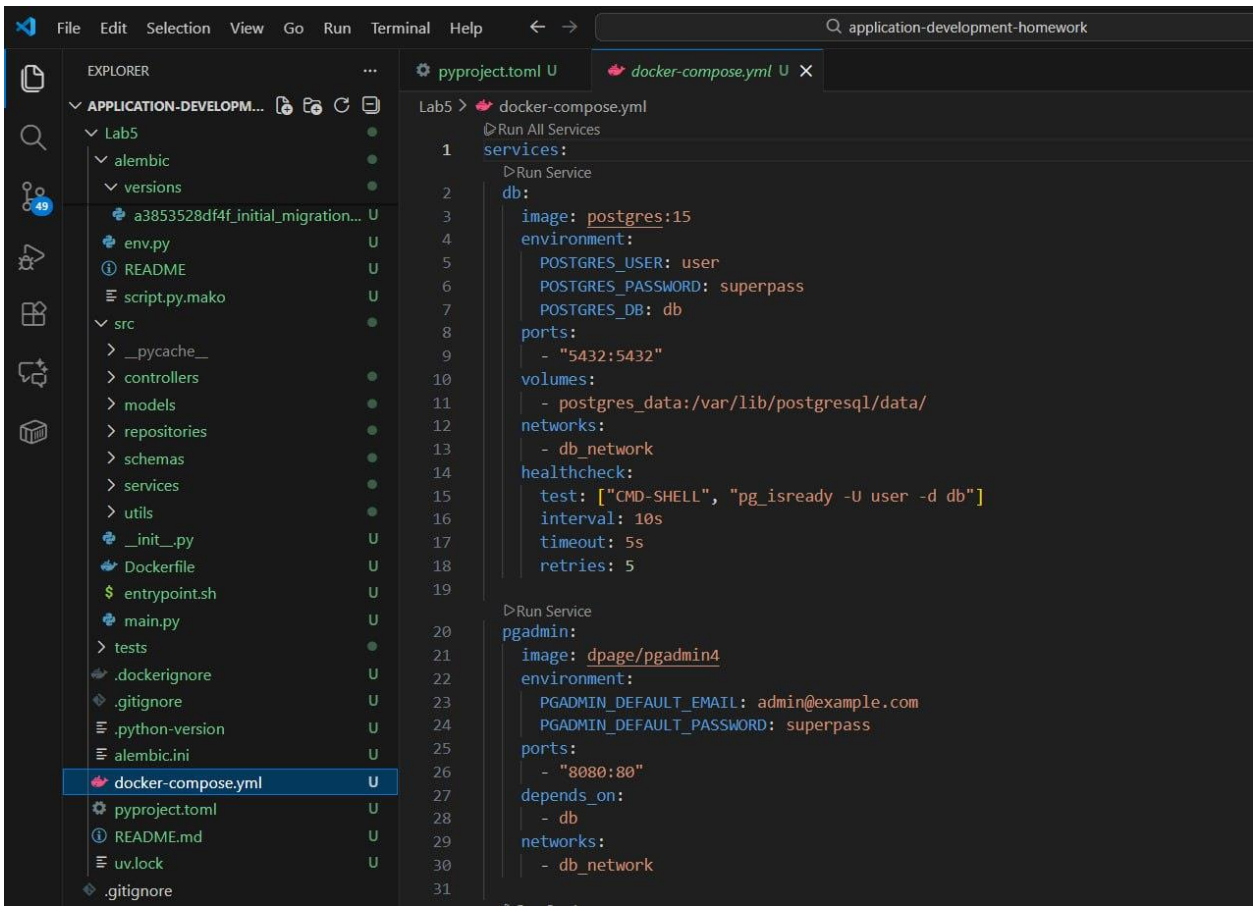
(скрипт запуска приложения, который выполняет критически важные операции перед стартом основного приложения.**)**

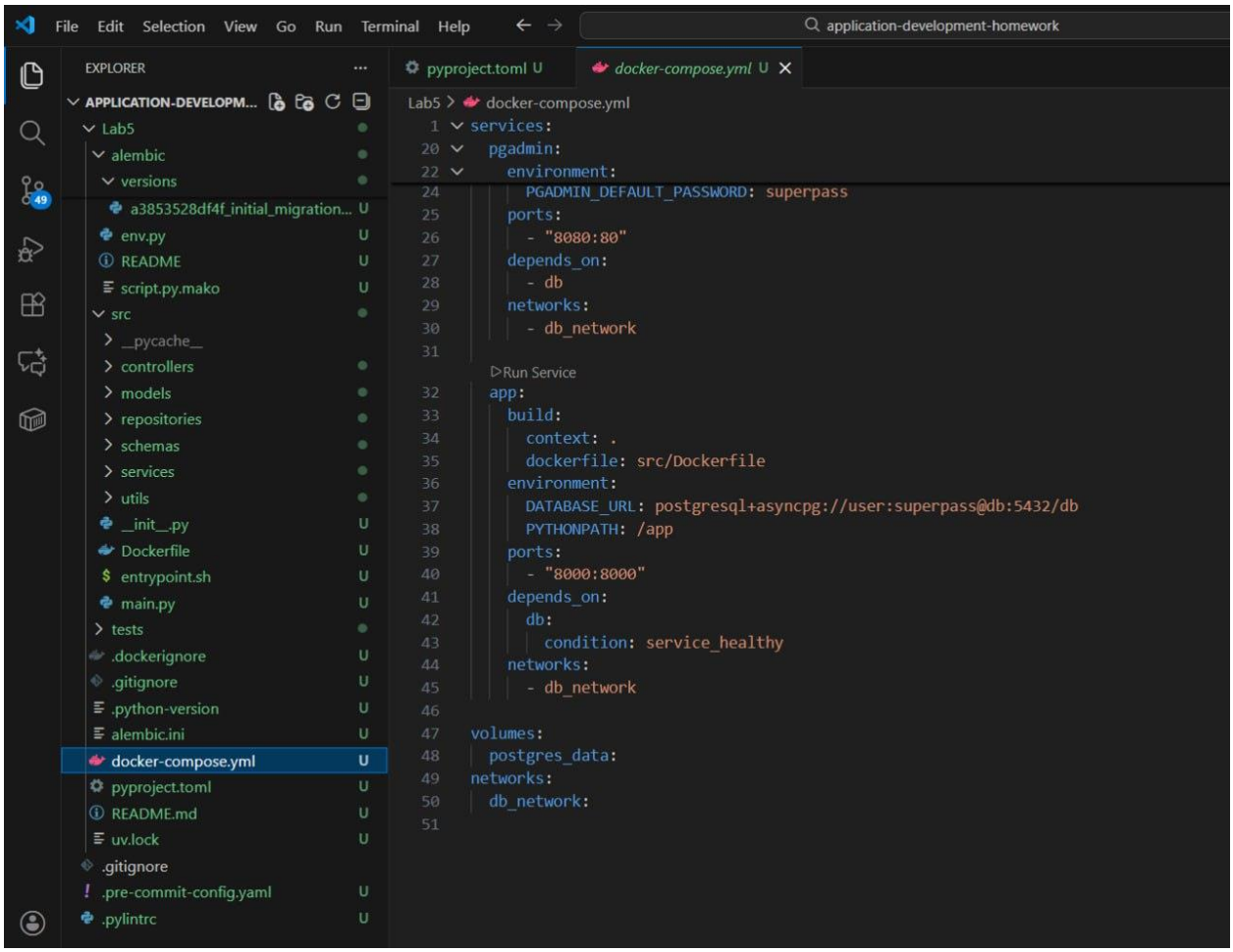


Описание работы:

- **Выполнение миграций базы данных при запуске контейнера:** команда запускает Alembic через uv для применения всех миграций до последней версии. Это гарантирует, что схема базы данных соответствует ожиданиям приложения
- После успешного выполнения миграций выводится сообщение о запуске приложения.

Файл: *docker-compose.yaml*


File Edit Selection View Go Run Terminal Help
application-development-homework
EXPLORER
APPLICATION-DEVELOPM...
Lab5
alembic
versions
a3853528df4f_initial_migration... U
env.py U
README U
script.py.mako U
src
__pycache__
controllers
models
repositories
schemas
services
utils
__init__.py U
Dockerfile U
entrypoint.sh U
main.py U
tests
.dockerignore U
.gitignore U
.python-version U
alembic.ini U
docker-compose.yml U
pyproject.toml U
README.md U
uv.lock U
.gitignore
pyproject.toml U
docker-compose.yml U
Lab5 > docker-compose.yml
Run All Services
services:
Run Service
db:
image: postgres:15
environment:
POSTGRES_USER: user
POSTGRES_PASSWORD: superpass
POSTGRES_DB: db
ports:
- "5432:5432"
volumes:
- postgres_data:/var/lib/postgresql/data/
networks:
- db_network
healthcheck:
test: ["CMD-SHELL", "pg_isready -U user -d db"]
interval: 10s
timeout: 5s
retries: 5
Run Service
pgadmin:
image: dpage/pgadmin4
environment:
PGADMIN_DEFAULT_EMAIL: admin@example.com
PGADMIN_DEFAULT_PASSWORD: superpass
ports:
- "8080:80"
depends_on:
- db
networks:
- db_network


File Edit Selection View Go Run Terminal Help
application-development-homework
EXPLORER
APPLICATION-DEVELOPM...
Lab5
alembic
versions
a3853528df4f_initial_migration... U
env.py U
README U
script.py.mako U
src
__pycache__
controllers
models
repositories
schemas
services
utils
__init__.py U
Dockerfile U
entrypoint.sh U
main.py U
tests
.dockerignore U
.gitignore U
.python-version U
alembic.ini U
docker-compose.yml U
pyproject.toml U
README.md U
uv.lock U
.gitignore
.pre-commit-config.yaml U
.pylintrc U
pyproject.toml U
docker-compose.yml U
Lab5 > docker-compose.yml
services:
pgadmin:
environment:
PGADMIN_DEFAULT_PASSWORD: superpass
ports:
- "8080:80"
depends_on:
- db
networks:
- db_network
Run Service
app:
build:
context: .
dockerfile: src/Dockerfile
environment:
DATABASE_URL: postgresql+asyncpg://user:superpass@db:5432/db
PYTHONPATH: /app
ports:
- "8000:8000"
depends_on:
db:
condition: service_healthy
networks:
- db_network
volumes:
postgres_data:
networks:
db_network:

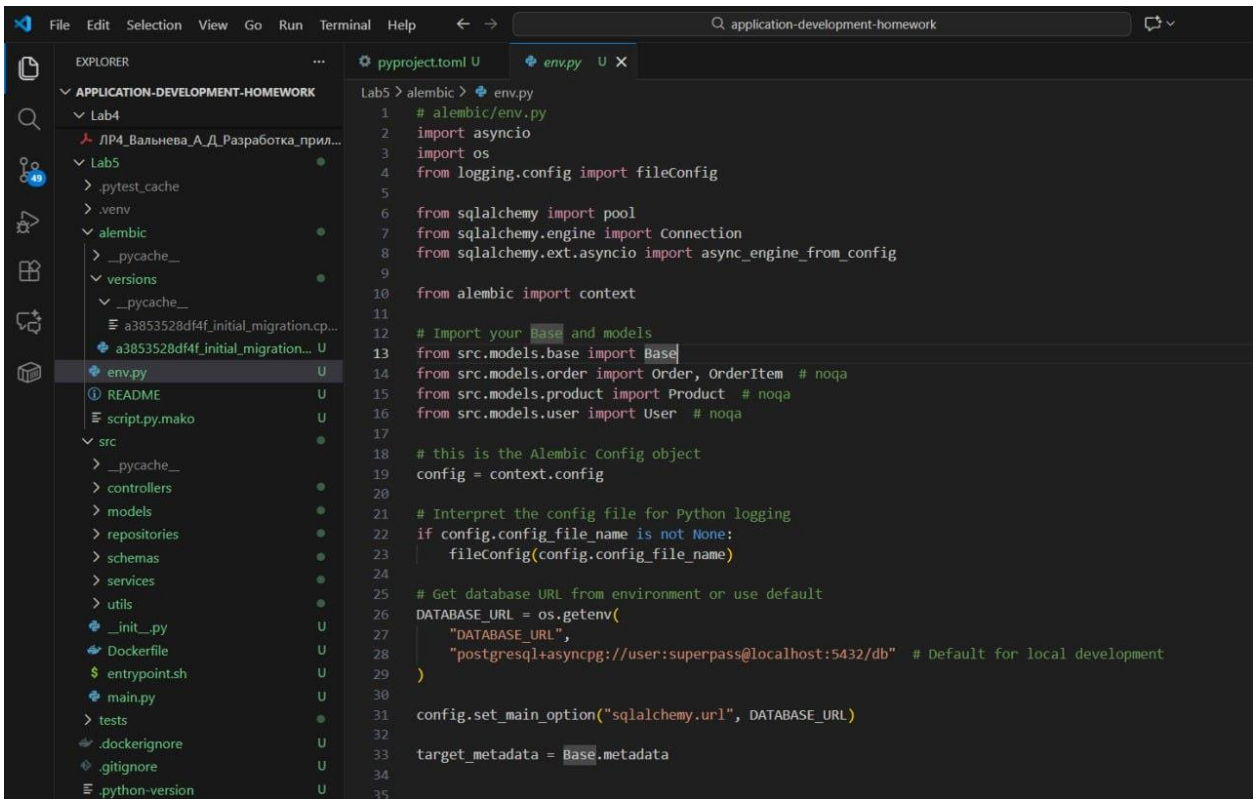
Была создана конфигурация **docker-compose.yaml** для оркестрации нескольких контейнеров.

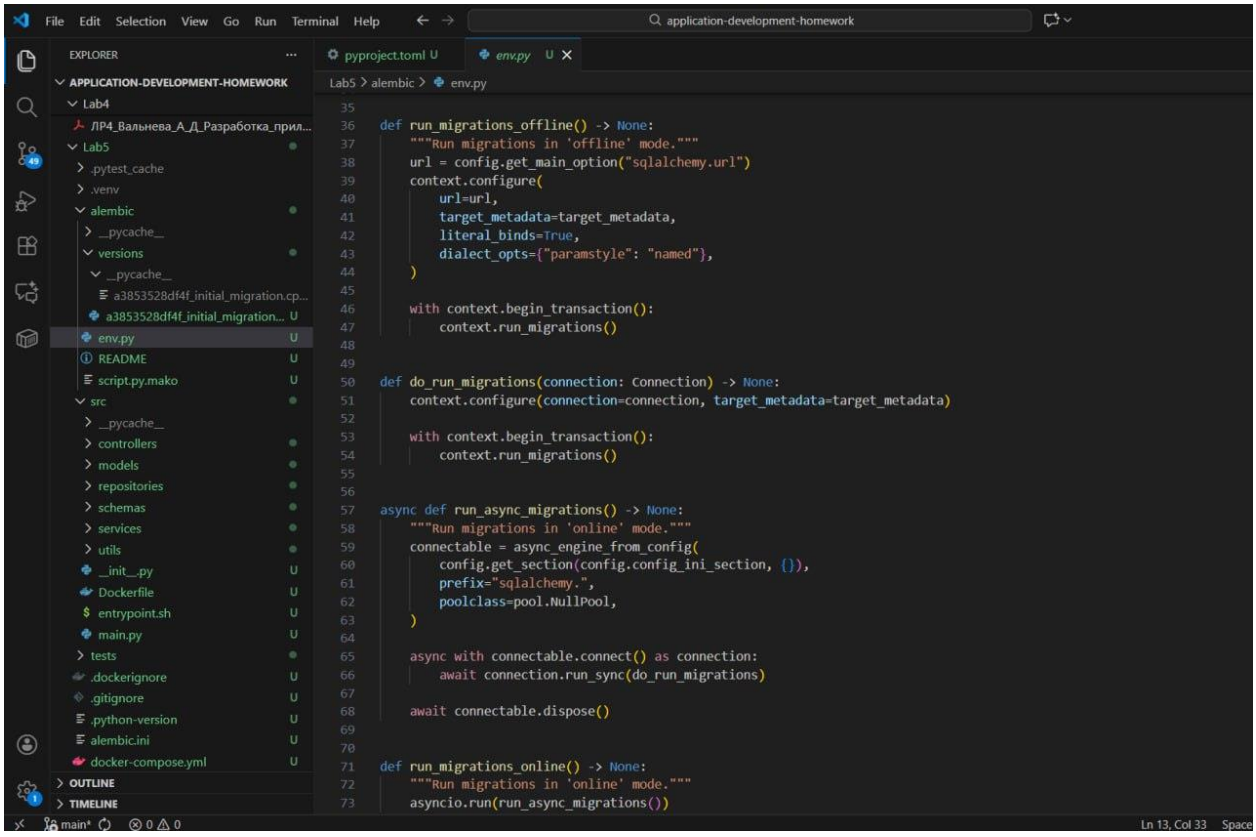
Описание работы конфигурации:

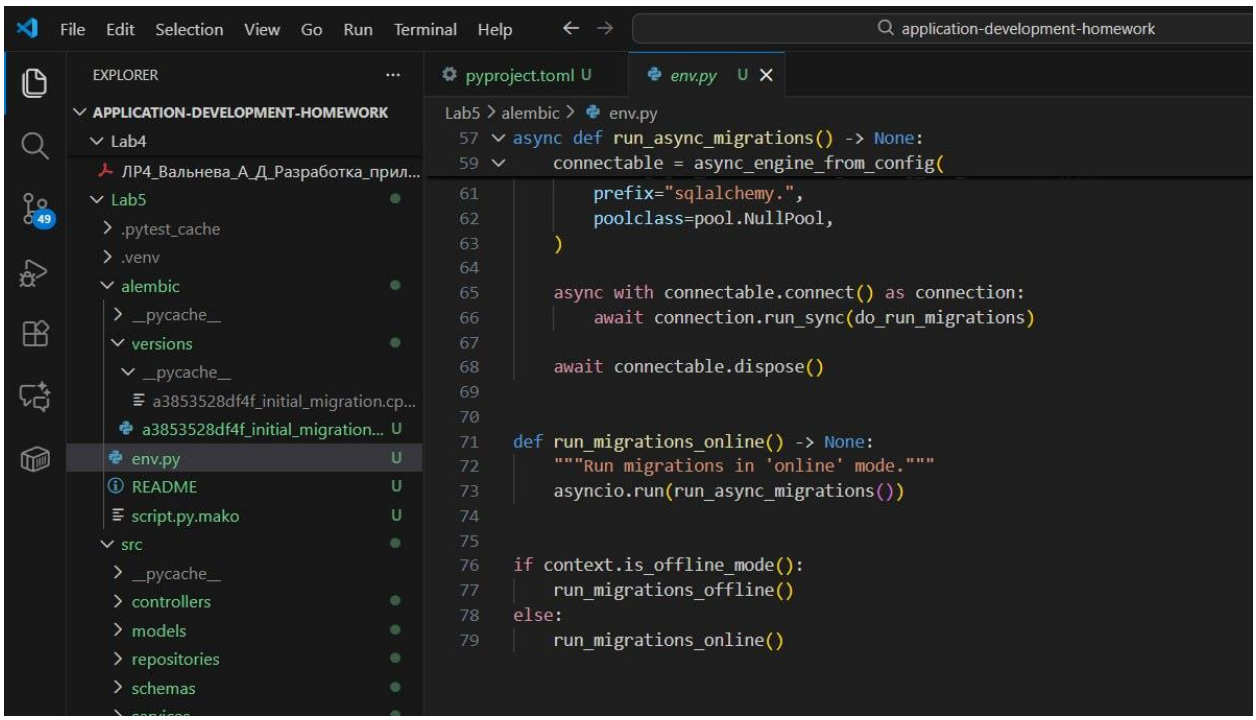
1. **Сервис приложения:**
   - Определен сервис для основного приложения
   - Указана зависимость от сервиса базы данных через параметр depends_on
   - Это гарантирует, что контейнер БД запустится раньше контейнера приложения
2. **Переменные окружения:**
   - Определены необходимые переменные для работы приложения:
     - DATABASE_URL - строка подключения к PostgreSQL
     - HOST и PORT - параметры для запуска веб-сервера
     - Другие переменные, специфичные для проекта
   - Переменные могут быть загружены из файла .env или определены непосредственно в docker-compose.yaml
3. **Сервис базы данных:**
   - Определен контейнер PostgreSQL
   - Настроены параметры подключения (имя пользователя, пароль, база данных)
   - Возможна настройка volume для персистентности данных
4. **Проброс портов:**
   - Порт 8000 приложения пробрасывается на хост для доступа к API

Часть 3: Интеграция с Alembic

Файл *env.py:*


File Edit Selection View Go Run Terminal Help
application-development-homework
EXPLORER
APPLICATION-DEVELOPMENT-HOMEWORK
Lab4
ЛР4_Вальнева_А_Д_Разработка_прил...
Lab5
.pytest_cache
.venv
alembic
__pycache__
versions
__pycache__
a3853528df4f_initial_migration.cp...
a3853528df4f_initial_migration... U
env.py U
README U
script.py.mako U
src
__pycache__
controllers
models
repositories
schemas
services
utils
__init__.py U
Dockerfile U
entrypoint.sh U
main.py U
tests
.dockerignore U
.gitignore U
.python-version U
pyproject.toml U
env.py U
Lab5 > alembic > env.py
# alembic/env.py
import asyncio
import os
from logging.config import fileConfig
from sqlalchemy import pool
from sqlalchemy.engine import Connection
from sqlalchemy.ext.asyncio import async_engine_from_config
from alembic import context
# Import your Base and models
from src.models.base import Base
from src.models.order import Order, OrderItem # noqa
from src.models.product import Product # noqa
from src.models.user import User # noqa
# this is the Alembic Config object
config = context.config
# Interpret the config file for Python logging
if config.config_file_name is not None:
fileConfig(config.config_file_name)
# Get database URL from environment or use default
DATABASE_URL = os.getenv(
"DATABASE_URL",
"postgresql+asyncpg://user:superpass@localhost:5432/db" # Default for local development
)
config.set_main_option("sqlalchemy.url", DATABASE_URL)
target_metadata = Base.metadata

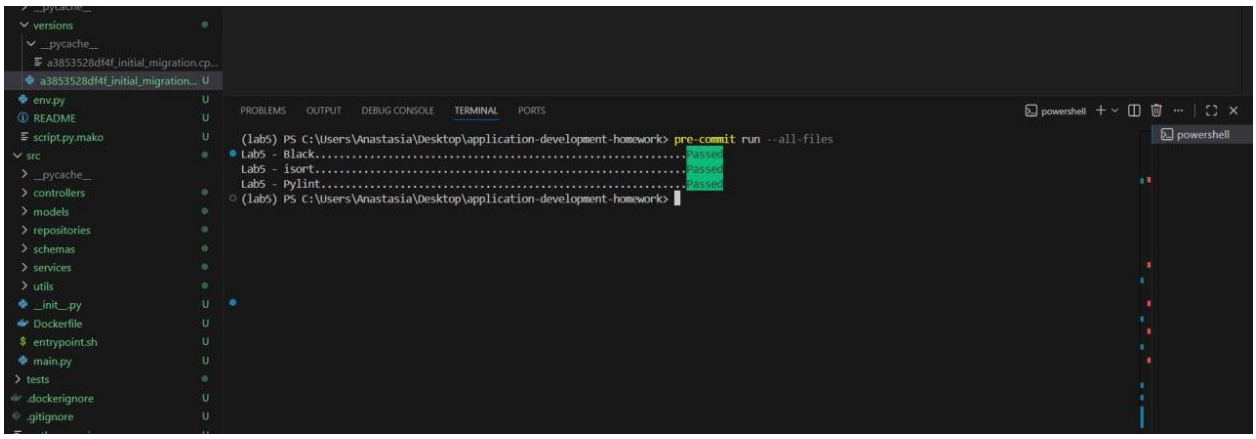

File Edit Selection View Go Run Terminal Help
application-development-homework
EXPLORER
APPLICATION-DEVELOPMENT-HOMEWORK
Lab4
ЛР4_Вальнева_А_Д_Разработка_прил...
Lab5
.pytest_cache
.venv
alembic
__pycache__
versions
__pycache__
a3853528df4f_initial_migration.cp...
a3853528df4f_initial_migration... U
env.py U
README U
script.py.mako U
src
__pycache__
controllers
models
repositories
schemas
services
utils
__init__.py U
Dockerfile U
entrypoint.sh U
main.py U
tests
.dockerignore U
.gitignore U
.python-version U
alembic.ini U
docker-compose.yml U
OUTLINE
TIMELINE
pyproject.toml U
env.py U
Lab5 > alembic > env.py
def run_migrations_offline() -> None:
"""Run migrations in 'offline' mode."""
url = config.get_main_option("sqlalchemy.url")
context.configure(
url=url,
target_metadata=target_metadata,
literal_binds=True,
dialect_opts={"paramstyle": "named"},
)
with context.begin_transaction():
context.run_migrations()
def do_run_migrations(connection: Connection) -> None:
context.configure(connection=connection, target_metadata=target_metadata)
with context.begin_transaction():
context.run_migrations()
async def run_async_migrations() -> None:
"""Run migrations in 'online' mode."""
connectable = async_engine_from_config(
config.get_section(config.config_ini_section, {}),
prefix="sqlalchemy.",
poolclass=pool.NullPool,
)
async with connectable.connect() as connection:
await connection.run_sync(do_run_migrations)
await connectable.dispose()
def run_migrations_online() -> None:
"""Run migrations in 'online' mode."""
asyncio.run(run_async_migrations())
main* 0 0
Ln 13, Col 33 Space



Этот файл конфигурирует работу Alembic в проекте:

- **Импорт моделей:** импортируются все модели данных и базовый класс Base
- **Получение DATABASE_URL:** URL базы данных читается из переменной окружения с fallback на локальное значение по умолчанию
- **Настройка метаданных:** target_metadata = Base.metadata указывает Alembic на метаданные SQLAlchemy для автогенерации миграций
- **Асинхронные миграции:** реализована функция run_async_migrations() для поддержки асинхронного драйвера asyncpg

Также автоматически создан файл миграции **a3853528df4f_initial_migration.py**: представляет собой начальную миграцию, которая создает всю структуру базы данных:



Процесс запуска:

1. Docker Compose собирает образ приложения
2. Запускается контейнер PostgreSQL
3. Запускается контейнер приложения
4. Выполняется entrypoint.sh:
    - Ожидание готовности БД
    - Применение миграций
    - Запуск приложения
5. Приложение становится доступным на порту 8000

Ответы на вопросы

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

Docker-контейнер — это изолированная среда для запуска приложений, которая содержит все необходимые зависимости, библиотеки и конфигурации, обеспечивая быстрое развертывание и воспроизводимость окружения.

Основные отличия от виртуальной машины:

1. Архитектура: контейнер использует общее ядро ОС хоста, в то время как виртуальная машина имеет собственное полноценное ядро операционной системы и работает через гипервизор.
2. Размер: контейнеры значительно легче (обычно десятки-сотни

мегабайт), виртуальные машины занимают гигабайты, так как содержат полную ОС.

3. Скорость запуска: контейнеры запускаются за секунды, виртуальные машины — за минуты, поскольку требуют загрузки полной операционной системы.
4. Изоляция: контейнеры обеспечивают изоляцию на уровне процессов, виртуальные машины — полную изоляцию на уровне аппаратного обеспечения.
5. Потребление ресурсов: контейнеры более эффективны и потребляют меньше ресурсов (CPU, RAM)

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

Docker образ состоит из множества слоев, каждый из которых создается отдельной инструкцией в Dockerfile (FROM, RUN, COPY и т.д.). Docker кеширует каждый слой и при повторной сборке использует кеш, если инструкция и контекст не изменились.

Также в Dockerfile сначала копируются и устанавливаются зависимости (requirements.txt или uv), и только потом копируется код приложения. Это позволяет при изменении кода не переустанавливать зависимости, так как соответствующий слой берется из кеша. Если бы порядок был обратным, при каждом изменении кода пришлось бы переустанавливать все пакеты.

Почему это важно:

1. **Скорость сборки:** использование кеша значительно ускоряет повторные сборки образа — вместо минут сборка может занимать секунды.
2. **Эффективность разработки:** при изменении только исходного кода приложения не нужно заново устанавливать все зависимости.
3. **Экономия ресурсов:** уменьшается нагрузка на процессор и сеть (не нужно повторно скачивать пакеты)

3. Что означает инструкция depends_on в docker-compose?

Инструкция depends_on определяет порядок запуска контейнеров и зависимости между сервисами. В docker-compose.yaml приложение имеет depends_on: db, что означает, что контейнер PostgreSQL (db) запустится раньше контейнера приложения, но PostgreSQL может быть еще не готов принимать соединения. Поэтому в entrypoint.sh добавлена дополнительная логика ожидания:

- bashuntil pg_isready -h db -p 5432 -U user > /dev/null 2>&1; do
  echo "PostgreSQL is unavailable - sleeping"
  sleep 2
  done

Так проверяем готовность базы данных и компенсируем ограничение depends_on

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

Причины выполнения миграций в entrypoint.sh:

1. Отсутствие доступа к БД при сборке: во время выполнения docker build база данных не запущена и недоступна. Миграции требуют активного подключения к БД, которое появляется только при запуске контейнеров через docker compose up.
2. Образ Docker должен быть универсальным и независимым от конкретной базы данных. Один и тот же образ может использоваться с разными БД (development, staging, production).
3. Безопасность: Учетные данные БД (пароли, хосты) не должны "зашиваться" в образ во время сборки. Они передаются через переменные окружения при запуске.
4. Актуальность схемы: Миграции должны выполняться каждый раз при запуске, чтобы схема БД соответствовала версии приложения. Если выполнить миграции при сборке, они станут устаревшими.

5. Что произойдет, если миграции завершатся ошибкой при запуске

контейнера?

При ошибке миграций:

1. Остановка запуска благодаря директиве set -e в начале entrypoint.sh, скрипт немедленно прервется при любой ошибке. Команда exec "$@" (запуск приложения) не выполнится.
2. Контейнер завершит работу, перейдет в состояние "Exited" с ненулевым кодом возврата, что сигнализирует об ошибке.
3. Приложение не запустится: Веб-сервер (uvicorn) не будет запущен, приложение останется недоступным.
4. В выводе логов (docker compose logs/ docker logs) будет видна ошибка миграции.

Если в файле миграции a3853528df4f_initial_migration.py есть ошибка, или база данных находится в несовместимом состоянии, контейнер остановится с сообщением об ошибке, что позволяет быстро локализовать и исправить проблему.

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

Форматеры (выполняются первыми): автоматически форматируют код, приводя его к определенному стилю.

- Выравнивают отступы
- Расставляют пробелы вокруг операторов
- Переносят длинные строки
- Упорядочивают импорты (isort)
- Нормализуют кавычки
- Удаляют лишние пробелы и пустые строки

Результат: Отформатированный код, полностью соответствующий стандарту форматера.

Линтеры (выполняются после): проводят статический анализ кода для поиска ошибок, потенциальных проблем, нарушений стиля и best practices.

- Обнаруживают синтаксические ошибки
- Находят неиспользуемые импорты и переменные
- Выявляют потенциально опасный код (например, изменяемые значения по умолчанию)
- Проверяют соблюдение соглашений о стиле (PEP 8)
- Обнаруживают слишком сложный код (высокая цикломатическая сложность)
- Предупреждают о нарушениях best practices

Анализируют и сообщают о проблемах. Не изменяют код.
Результат: Список предупреждений и ошибок с указанием места и типа проблемы.

7. Как pre-commit хуки помогают в разработке?

Pre-commit хуки — это скрипты, которые автоматически выполняются перед созданием коммита в Git.

Как помогают в разработке:

1. Автоматически проверяю код на соответствие стандартам: невозможно закоммитить код с ошибками форматирования; проблемы обнаруживаются немедленно.
2. Экономят время команды на форматирование кода, в т.ч. не нужно вручную запускать линтеры перед каждым коммитом
3. Неиспользуемые импорты удаляются автоматически
4. Потенциальные проблемы выявляются на ранней стадии

Вывод:

В результате выполнения лабораторной работы №5 были изучены и настроены инструменты контроля качества кода: pre-commit хуки для автоматической проверки перед коммитами; black для автоматического форматирования; isort для упорядочивания импортов; pylint с детальной конфигурацией

для статического анализа, а также инструменты контейнеризации приложения: dockerfile для сборки образа приложения; скрипт entrypoint.sh для правильной инициализации; Docker Compose для оркестрации контейнеров; уастройка миграций базы данных при запуске

Все компоненты интегрированы и работают корректно, что обеспечивает поддержание высокого качества кода и упрощает развертывание приложения в различных окружениях/