

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.

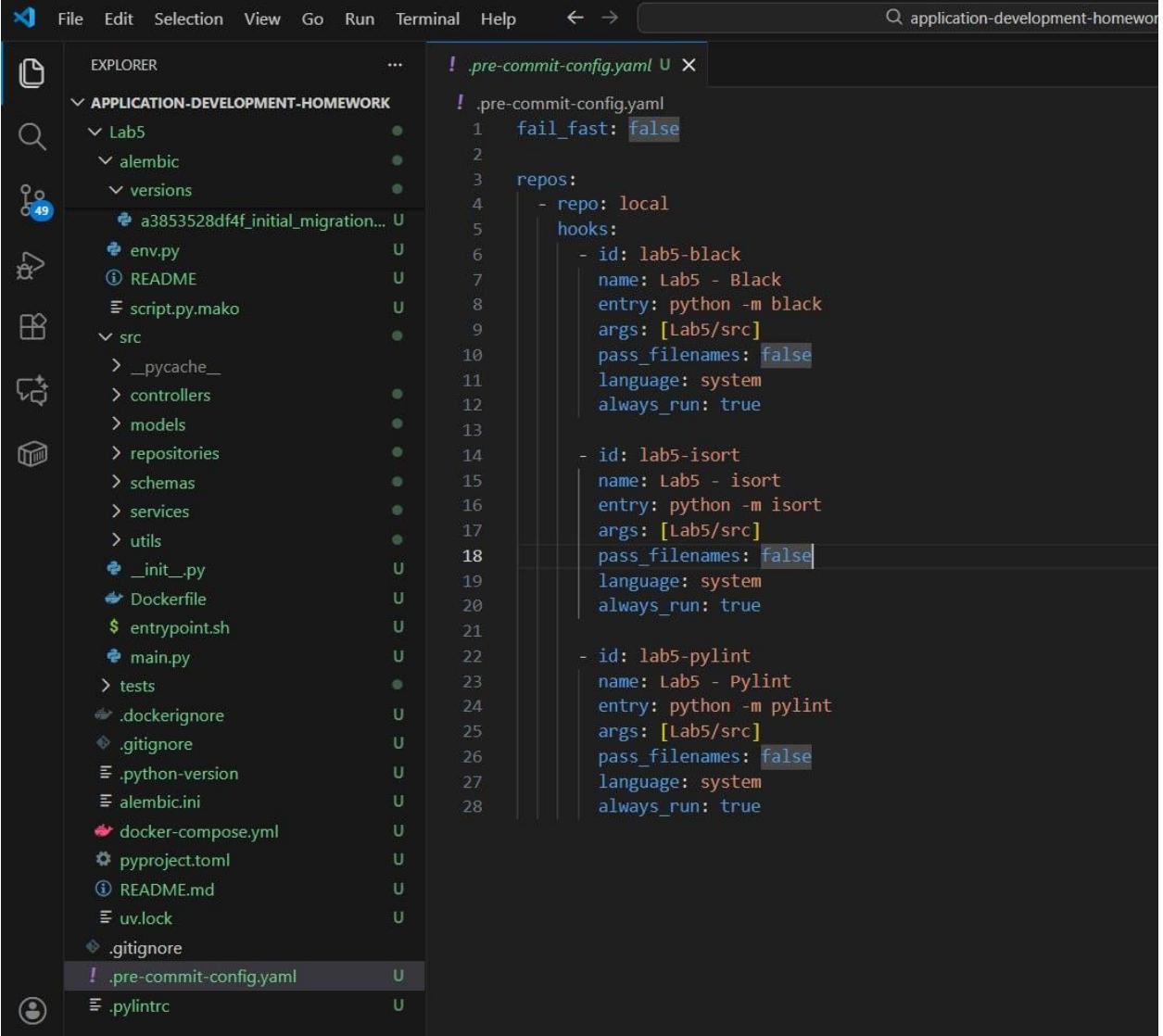
Задачи

- 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



The screenshot shows a code editor interface with a dark theme. On the left is the Explorer sidebar, which lists a project structure under 'APPLICATION-DEVELOPMENT-HOMEWORK'. The structure includes a 'Lab5' folder containing 'alembic' and 'src' subfolders, along with files like 'env.py', 'README', 'script.py.mako', 'Dockerfile', 'entrypoint.sh', 'main.py', 'tests', '.dockercfgignore', '.gitignore', '.python-version', 'alembic.ini', 'docker-compose.yml', 'pyproject.toml', 'README.md', 'uv.lock', and '.gitignore'. The current file being edited is '.pre-commit-config.yaml' in the main editor area. The code content is as follows:

```
! .pre-commit-config.yaml U X
!
! .pre-commit-config.yaml
1 fail_fast: false
2
3 repos:
4   - repo: local
5     hooks:
6       - id: lab5-black
7         name: Lab5 - Black
8         entry: python -m black
9         args: [Lab5/src]
10        pass_filenames: false
11        language: system
12        always_run: true
13
14   - id: lab5-isort
15     name: Lab5 - isort
16     entry: python -m isort
17     args: [Lab5/src]
18     pass_filenames: false
19     language: system
20     always_run: true
21
22   - id: lab5-pylint
23     name: Lab5 - Pylint
24     entry: python -m pylint
25     args: [Lab5/src]
26     pass_filenames: false
27     language: system
28     always_run: true
```

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)

The screenshot shows the VS Code interface with the title bar "application-development-homework". The left sidebar displays a file tree for "APPLICATION-DEVELOPMENT-HOMEWORK" containing various files like Lab5, alembic, versions, src, tests, Dockerfile, and pyproject.toml. The main editor area shows the contents of the ".pylintrc" configuration file.

```
1 [MASTER]
2 # Add the root directory to Python path for imports
3 init-hook='import sys; import os; sys.path.insert(0, os.path.dirname(os.path.abspath(".")))'
4
5 # Ignore paths
6 ignore-paths=
7   ^deploy/*$,
8   ^temp/*$,
9   ^build/*$,
10  ^tests/*$,
11  ^.*/alembic/*$
12
13 [MESSAGES CONTROL]
14 # Disable specific checks that are acceptable for this project
15 disable=
16   missing-function-docstring,
17   missing-module-docstring,
18   missing-class-docstring,
19   too-few-public-methods,
20   too-many-arguments,
21   too-many-positional-arguments,
22   too-many-instance-attributes,
23   import-outside-toplevel,
24   duplicate-code,
25   redefined-outer-name,
26   unused-argument,
27   broad-exception-caught
28
29 [TYPECHECK]
30 # SQLAlchemy and Pydantic use metaclass magic that Pylint doesn't understand
31 generated-members=
32   func.now,
33   func.count,
34   func.*,
35   select.*,
36   insert.*,
37   update.*,
38   delete.*,
39   sqlalchemy.*
40
41 extension-pkg-whitelist=
42   pydantic,
43   orjson,
44   sqlalchemy
45
46 # Ignore "not callable" errors for SQLAlchemy func
47 ignored-classes=
48   sqlalchemy.sql.functions.func,
49   sqlalchemy.orm.scoped_session
50
51 [DESIGN]
52 # Reduce minimum public methods requirement for data models
53 min-public-methods=0
54 max-attributes=10
55 max-args=8
56
57 [FORMAT]
58 # Match Black's formatting
59 max-line-length=88
```

This screenshot shows the same VS Code interface as the first one, but the ".pylintrc" file contains more configuration sections. The "disabled" section has been added, and new sections for "TYPECHECK", "DESIGN", and "FORMAT" have been included.

```
1.0 disable=
28
29 [TYPECHECK]
30 # SQLAlchemy and Pydantic use metaclass magic that Pylint doesn't understand
31 generated-members=
32   func.now,
33   func.count,
34   func.*,
35   select.*,
36   insert.*,
37   update.*,
38   delete.*,
39   sqlalchemy.*
40
41 extension-pkg-whitelist=
42   pydantic,
43   orjson,
44   sqlalchemy
45
46 # Ignore "not callable" errors for SQLAlchemy func
47 ignored-classes=
48   sqlalchemy.sql.functions.func,
49   sqlalchemy.orm.scoped_session
50
51 [DESIGN]
52 # Reduce minimum public methods requirement for data models
53 min-public-methods=0
54 max-attributes=10
55 max-args=8
56
57 [FORMAT]
58 # Match Black's formatting
59 max-line-length=88
```

The screenshot shows the Visual Studio Code interface. The left sidebar displays a project structure for 'APPLICATION-DEVELOPMENT-HOMEWORK' containing files like 'Lab5', 'alembic', 'versions', 'src', and various Python scripts. The right side shows the content of the '.pylintrc' configuration file.

```
47  # ignore classes defined in sqlalchemy.orm.scoping.scoped_session
48
49  [DESIGN]
50
51  # Reduce minimum public methods requirement for data models
52  min-public-methods=0
53
54  max-attributes=10
55
56  max-args=8
57
58  [FORMAT]
59  # Match Black's formatting
60  max-line-length=88
61
62  [BASIC]
63  # Naming conventions
64  good-names=
65    i,j,k,
66    ex,
67    ,
68    id,
69    db,
70    app,
71    e
72
73  [SIMILARITIES]
74  # Minimum lines of similar code to report
75  min-similarity-lines=10
```

Описание конфигурации:

Секция [MASTER]:

- init-hook: код выполняется перед запуском Pylint, 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

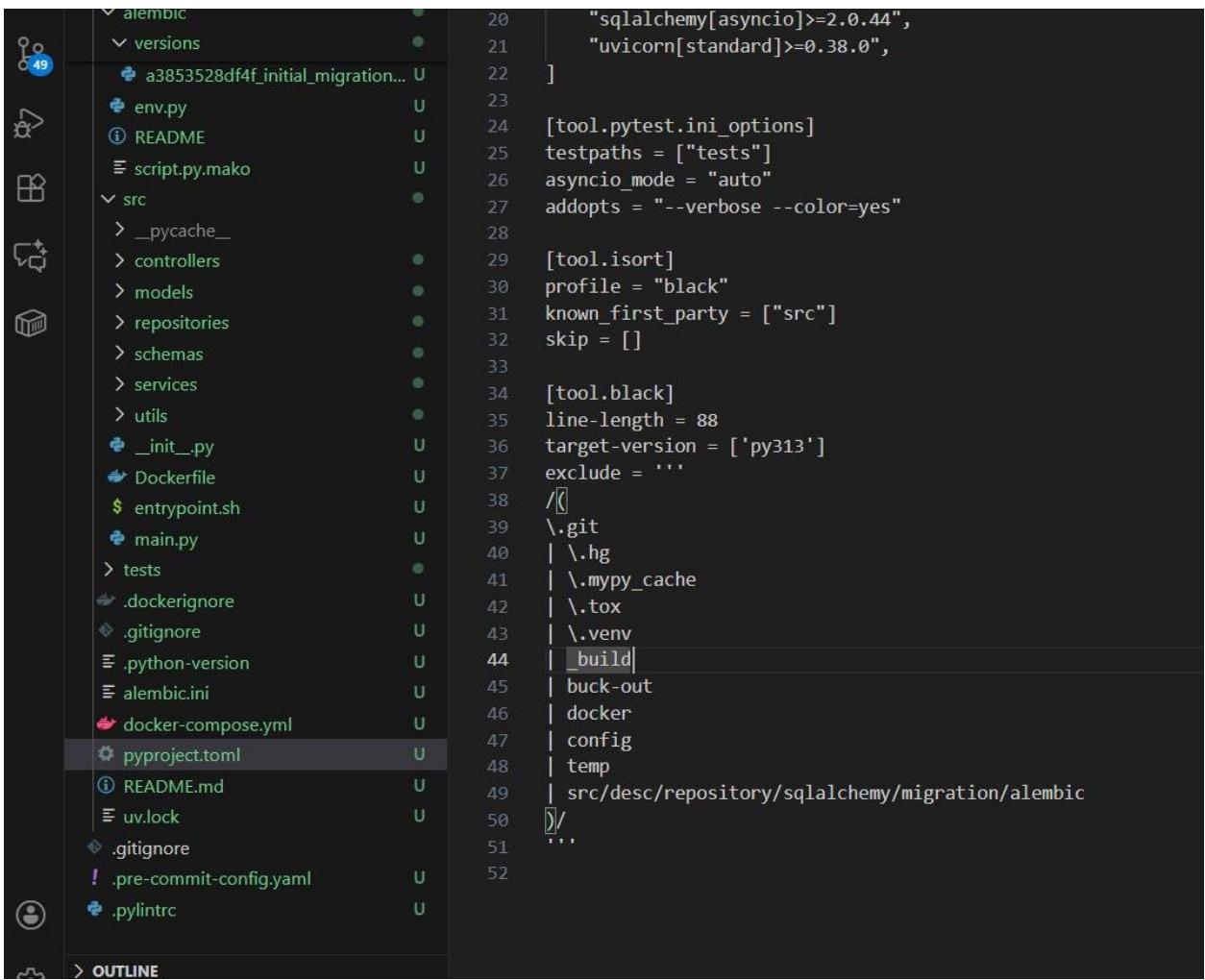
Установка pre-commit хуков (pre-commit install активирует автоматические проверки перед каждым коммитом)

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*



```
20     "sqlalchemy[asyncio]>=2.0.44",
21     "uvicorn[standard]>=0.38.0",
22   ]
23
24 [tool.pytest.ini_options]
25 testpaths = ["tests"]
26 asyncio_mode = "auto"
27 adopts = "--verbose --color=yes"
28
29 [tool.isort]
30 profile = "black"
31 known_first_party = ["src"]
32 skip = []
33
34 [tool.black]
35 line-length = 88
36 target-version = ['py313']
37 exclude = ''
38 /[]
39 \.git
40 | \.hg
41 | \.mypy_cache
42 | \.tox
43 | \.venv
44 | build
45 | buck-out
46 | docker
47 | config
48 | temp
49 | src/desc/repository/sqlalchemy/migration/alembic
50 | ...
51 ...
52 ...
```

Добавлены конфигурации

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

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

The screenshot shows a code editor interface with a sidebar and a main editor area. The sidebar on the left lists files and folders under 'EXPLORER'. The main editor area shows a file named 'Dockerfile' with the following content:

```
Lab5 > src > Dockerfile > FROM
1  FROM python:3.13-slim-trixie
2  COPY --from=ghcr.io/astral-sh/uv:latest /uv /uvx /bin/
3
4  WORKDIR /app
5
6  RUN apt-get update && apt-get install -y \
7      gcc \
8      postgresql-client \
9      curl \
10     && rm -rf /var/lib/apt/lists/*
11
12 COPY pyproject.toml uv.lock* ./
13
14 RUN mkdir -p /app/src
15 COPY src/ /app/src
16
17 COPY alembic.ini /app
18 COPY alembic/ /app/alembic
19
20 RUN uv sync --frozen --all-extras
21
22 RUN chmod +x /app/src/entrypoint.sh
23
24 ENV PYTHONPATH=/app/src
25
26 EXPOSE 8000
27
28 ENTRYPOINT ["/app/src/entrypoint.sh"]
29
30 CMD ["uv", "run", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
31
```

Файл: *entrypoint.sh*

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

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a project structure under "APPLICATION-DEVELOPMENT-HOMEWORK".
 - Lab5 folder contains alembic and versions subfolders.
 - src folder contains __pycache__, controllers, models, repositories, schemas, services, utils, __init__.py, Dockerfile, and entrypoint.sh.
 - entrypoint.sh is selected in the file list.
- Terminal:** Shows the content of entrypoint.sh:

```
#!/bin/bash
set -e
echo "Waiting for PostgreSQL to be ready..."
until pg_isready -h db -p 5432 -U user > /dev/null 2>&1; do
    echo "PostgreSQL is unavailable - sleeping"
    sleep 2
done

echo "PostgreSQL is up - running migrations..."
cd /app

echo "Running migrations.."
uv run alembic upgrade head

echo "Starting application..."
cd /app/src
exec "$@"
```

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

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows the project structure under "APPLICATION-DEVELOPM...".
- Editor:** Displays the contents of the `docker-compose.yml` file.
- Terminal:** Shows the command `application-development-homework`.

```
version: '3.8'
services:
  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
  pgadmin:
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: superpass
    ports:
      - "8080:80"
    depends_on:
      - db
    networks:
      - db_network
```

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows the project structure under "APPLICATION-DEVELOPM...".
- Editor:** Displays the contents of the `docker-compose.yml` file with syntax highlighting.
- Terminal:** Shows the command `application-development-homework`.

```
version: '3.8'
services:
  db:
    image: postgres:15
    environment:
      PGADMIN_DEFAULT_PASSWORD: superpass
    ports:
      - "8080:80"
    depends_on:
      - db
    networks:
      - db_network
  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`:

The screenshot shows a code editor interface with a dark theme. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. A search bar at the top right contains the text "application-development-homework". The left sidebar is titled "EXPLORER" and shows a project structure under "APPLICATION-DEVELOPMENT-HOMEWORK". The "alembic" directory contains several files: __pycache__, versions, __pycache__, a3853528df4f_initial_migration.py, env.py, README, script.py.mako, Dockerfile, entrypoint.sh, main.py, tests, .dockerignore, .gitignore, and .python-version. The "src" directory contains __pycache__, controllers, models, repositories, schemas, services, and utils. The "env.py" file is selected and open in the main editor area.

```
Lab5 > alembic > env.py U
1 # alembic/env.py
2 import asyncio
3 import os
4 from logging.config import fileConfig
5
6 from sqlalchemy import pool
7 from sqlalchemy.engine import Connection
8 from sqlalchemy.ext.asyncio import async_engine_from_config
9
10 from alembic import context
11
12 # Import your Base and models
13 from src.models.base import Base
14 from src.models.order import Order, OrderItem # noqa
15 from src.models.product import Product # noqa
16 from src.models.user import User # noqa
17
18 # this is the Alembic Config object
19 config = context.config
20
21 # Interpret the config file for Python logging
22 if config.config_file_name is not None:
23     fileConfig(config.config_file_name)
24
25 # Get database URL from environment or use default
26 DATABASE_URL = os.getenv(
27     "DATABASE_URL",
28     "postgresql+asyncpg://user:superpass@localhost:5432/db" # Default for local development
29 )
30
31 config.set_main_option("sqlalchemy.url", DATABASE_URL)
32
33 target_metadata = Base.metadata
34
35
```

This screenshot shows the same code editor interface as the first one, but the "env.py" file has been modified. The changes are indicated by red highlights and numbers on the left margin. The modifications include:

- Line 35: Added docstring for run_migrations_offline.
- Line 36: Added code to run migrations in offline mode.
- Line 37: Set url to config.get_main_option("sqlalchemy.url").
- Line 38: Configure context with url and target_metadata.
- Line 39: Set literal_binds to True.
- Line 40: Set dialect_opts to {"paramstyle": "named"}.
- Line 41: Start a transaction and run migrations.
- Line 42: Added do_run_migrations function.
- Line 43: Configure context with connection and target_metadata.
- Line 44: Start a transaction and run migrations.
- Line 45: Added run_async_migrations function.
- Line 46: Set connectable to async_engine_from_config.
- Line 47: Set config.get_section to config.config_ini_section.
- Line 48: Set prefix to "sqlalchemy.".
- Line 49: Set poolclass to pool.NullPool.
- Line 50: Use connectable.connect to get a connection.
- Line 51: Await connection.run_sync(do_run_migrations).
- Line 52: Dispose of the connectable.
- Line 53: Added run_migrations_online function.
- Line 54: Set connectable to async_engine_from_config.
- Line 55: Set config.get_section to config.config_ini_section.
- Line 56: Set prefix to "sqlalchemy.".
- Line 57: Set poolclass to pool.NullPool.
- Line 58: Use connectable.connect to get a connection.
- Line 59: Await connection.run_sync(run_async_migrations).
- Line 60: Dispose of the connectable.

```
Lab5 > alembic > env.py
35 def run_migrations_offline() -> None:
36     """Run migrations in 'offline' mode.
37     url = config.get_main_option("sqlalchemy.url")
38     context.configure(
39         url=url,
40         target_metadata=target_metadata,
41         literal_binds=True,
42         dialect_opts={"paramstyle": "named"},
43     )
44
45     with context.begin_transaction():
46         context.run_migrations()
47
48 def do_run_migrations(connection: Connection) -> None:
49     context.configure(connection=connection, target_metadata=target_metadata)
50
51     with context.begin_transaction():
52         context.run_migrations()
53
54     async def run_async_migrations() -> None:
55         """Run migrations in 'online' mode.
56         connectable = async_engine_from_config(
57             config.get_section(config.config_ini_section, {}),
58             prefix="sqlalchemy.",
59             poolclass=pool.NullPool,
60         )
61
62         with connectable.connect() as connection:
63             await connection.run_sync(do_run_migrations())
64
65         await connectable.dispose()
66
67     def run_migrations_online() -> None:
68         """Run migrations in 'online' mode.
69         """
70         asyncio.run(run_async_migrations())
71
72
73
```

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows the project structure under 'APPLICATION-DEVELOPMENT-HOMEWORK'. It includes subfolders 'Lab4' and 'Lab5', and a 'src' folder containing 'controllers', 'models', 'repositories', 'schemas', and 'services'.
- Editor:** Displays the file 'env.py' with the following code:

```
Lab5 > alembic > env.py
57     def run_async_migrations() -> None:
59         connectable = async_engine_from_config(
61             prefix="sqlalchemy.",
62             poolclass=pool.NullPool,
63         )
64
65         async with connectable.connect() as connection:
66             await connection.run_sync(do_run_migrations)
67
68         await connectable.dispose()
69
70
71     def run_migrations_online() -> None:
72         """Run migrations in 'online' mode.
73         """
74         asyncio.run(run_async_migrations())
75
76     if context.is_offline_mode():
77         run_migrations_offline()
78     else:
79         run_migrations_online()
```

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:

```
(lab5) PS C:\Users\Anastasia\Desktop\application-development-homework> pre-commit run --all-files
● Lab5 - Black.....Passed
● Lab5 - isort.....Passed
○ Lab5 - Pylint.....Passed
```

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 обнаружаются 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/