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 №8:
Основы работы с планировщиками

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

Екатеринбург 2025

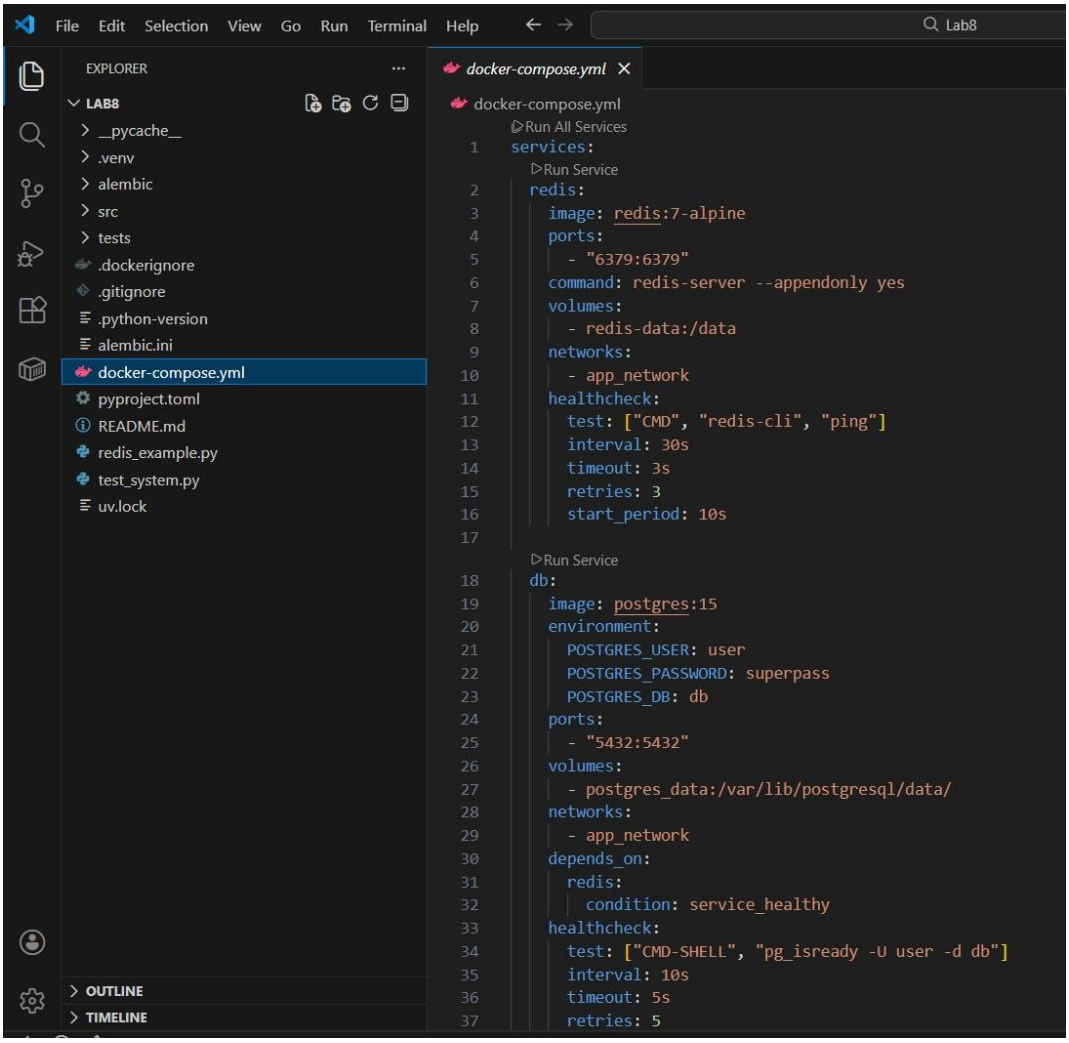
Цель работы

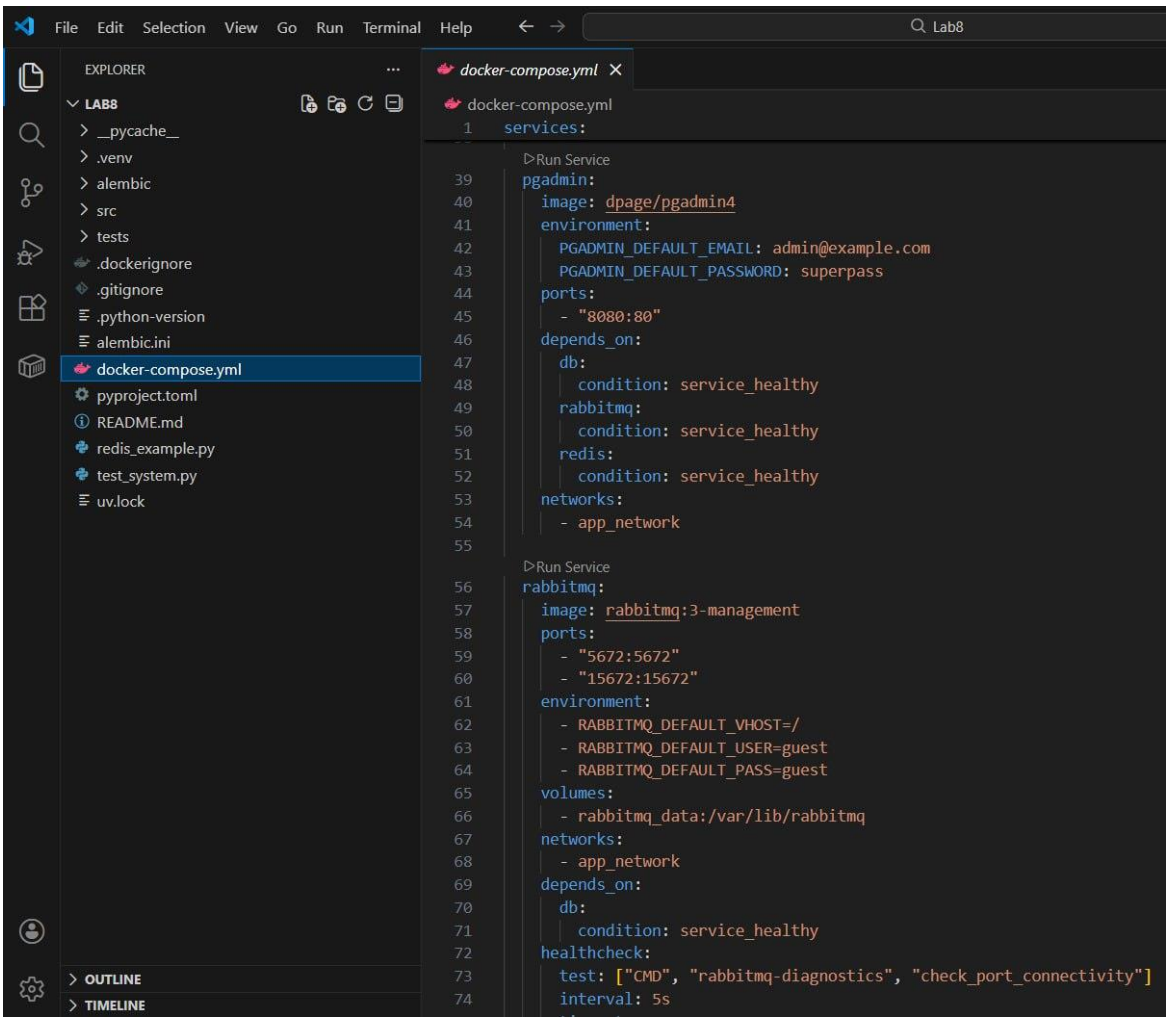
Научиться создавать, запускать и управлять асинхронными и отложенными задачами с помощью TaskIQ. Освоить работу с планировщиками задач для автоматической генерации отчетов.

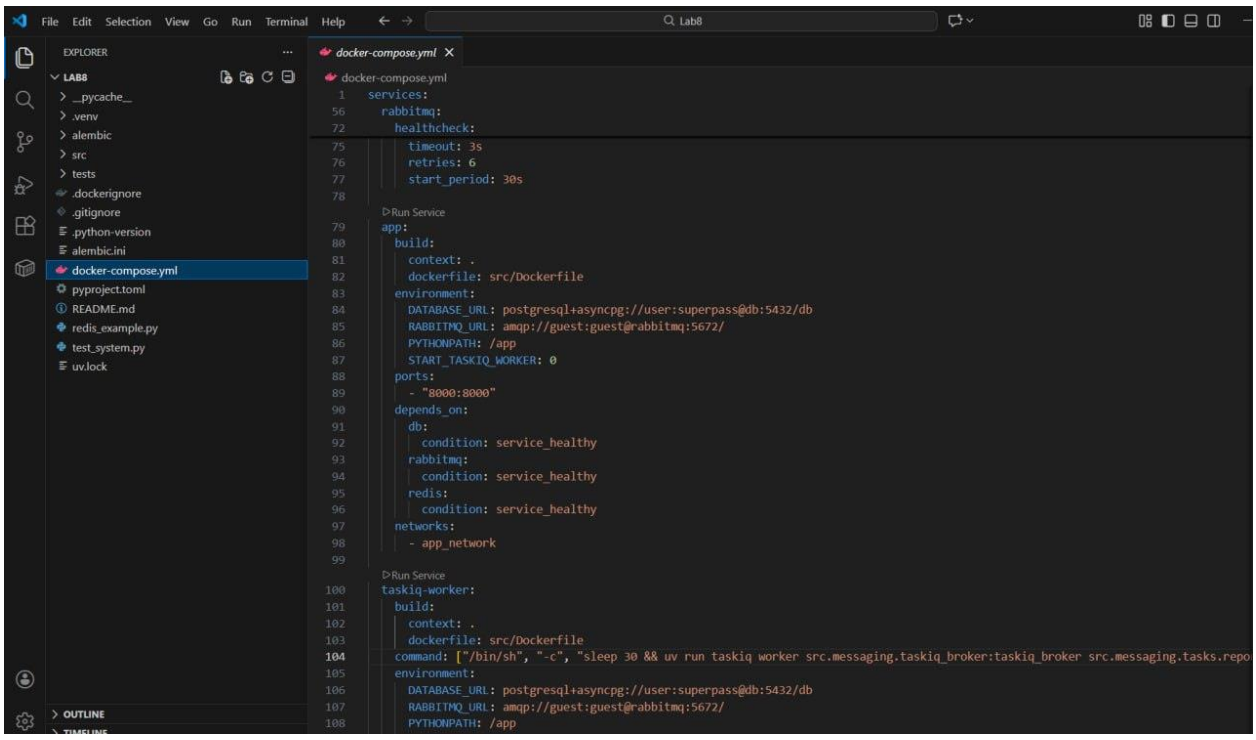
Задачи

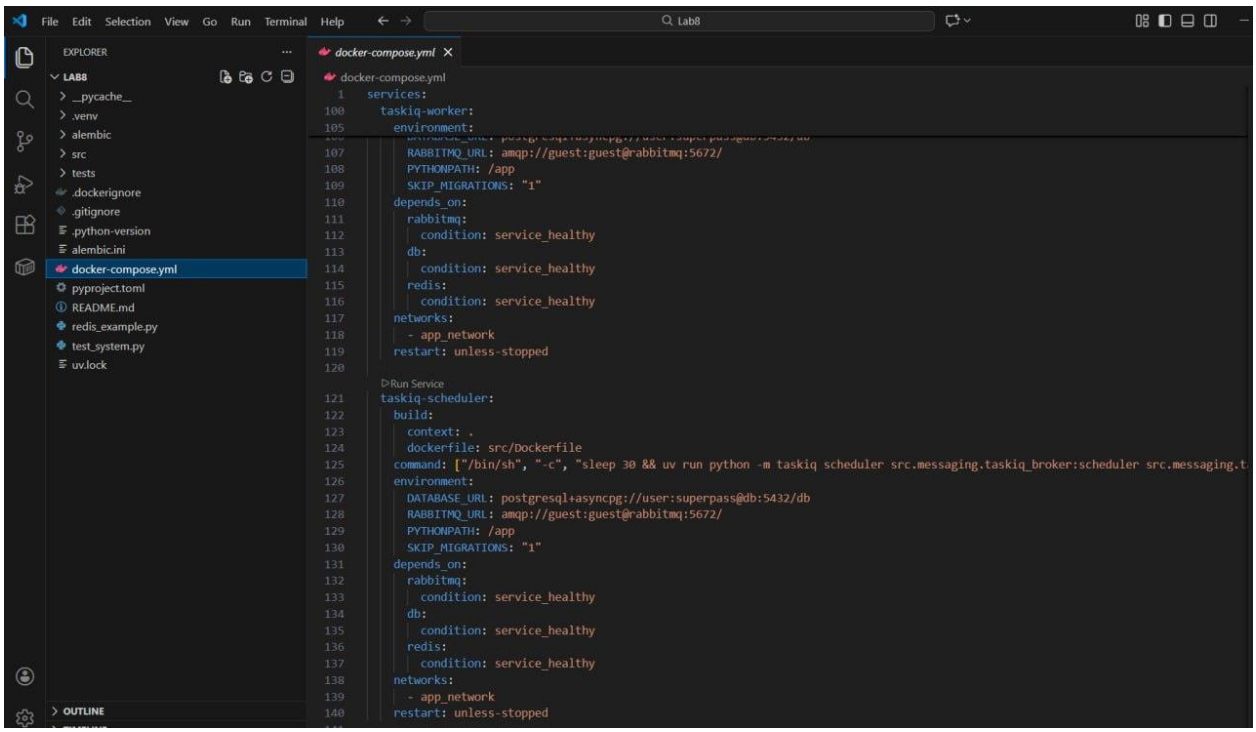
1. Создать представление (VIEW) в базе данных для формирования отчетов по заказам
2. Добавить миграцию для создания таблицы отчетов
3. Настроить планировщик TaskIQ для автоматической генерации отчетов
4. Создать REST API эндпоинт /report для получения отчетов
5. Реализовать автоматическую задачу, выполняющуюся по расписанию cron

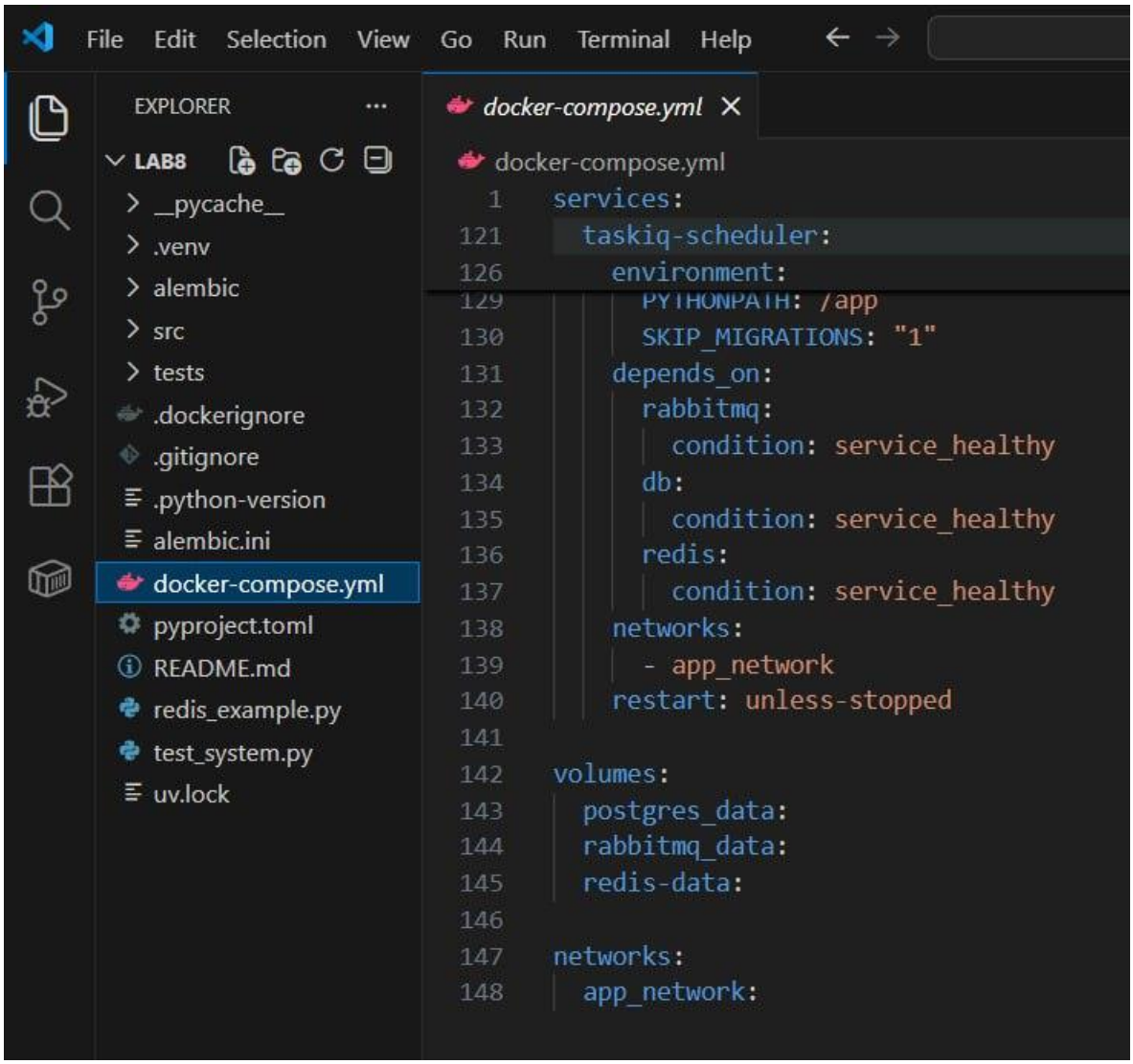
Часть 0: Настройка конфигурации Docker




File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
__pycache__
.venv
alembic
src
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
redis_example.py
test_system.py
uv.lock
OUTLINE
TIMELINE
docker-compose.yml
docker-compose.yml
services:
Run Service
pgadmin:
image: dpage/pgadmin4
environment:
PGADMIN_DEFAULT_EMAIL: admin@example.com
PGADMIN_DEFAULT_PASSWORD: superpass
ports:
- "8080:80"
depends_on:
db:
condition: service_healthy
rabbitmq:
condition: service_healthy
redis:
condition: service_healthy
networks:
- app_network
Run Service
rabbitmq:
image: rabbitmq:3-management
ports:
- "5672:5672"
- "15672:15672"
environment:
- RABBITMQ_DEFAULT_VHOST=/
- RABBITMQ_DEFAULT_USER=guest
- RABBITMQ_DEFAULT_PASS=guest
volumes:
- rabbitmq_data:/var/lib/rabbitmq
networks:
- app_network
depends_on:
db:
condition: service_healthy
healthcheck:
test: ["CMD", "rabbitmq-diagnostics", "check_port_connectivity"]
interval: 5s


File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
__pycache__
.venv
alembic
src
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
redis_example.py
test_system.py
uv.lock
OUTLINE
docker-compose.yml
services:
rabbitmq:
healthcheck:
timeout: 3s
retries: 6
start_period: 30s
Run Service
app:
build:
context: .
dockerfile: src/Dockerfile
environment:
DATABASE_URL: postgresql+asyncpg://user:superpass@db:5432/db
RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
PYTHONPATH: /app
START_TASKIQ_WORKER: 0
ports:
- "8000:8000"
depends_on:
db:
condition: service_healthy
rabbitmq:
condition: service_healthy
redis:
condition: service_healthy
networks:
- app_network
Run Service
taskiq-worker:
build:
context: .
dockerfile: src/Dockerfile
command: ["/bin/sh", "-c", "sleep 30 && uv run taskiq worker src.messaging.taskiq_broker:taskiq_broker src.messaging.tasks.repo
environment:
DATABASE_URL: postgresql+asyncpg://user:superpass@db:5432/db
RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
PYTHONPATH: /app


File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
pycache
.venv
alembic
src
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
redis_example.py
test_system.py
uv.lock
OUTLINE
docker-compose.yml
services:
taskiq-worker:
environment:
RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
PYTHONPATH: /app
SKIP_MIGRATIONS: "1"
depends_on:
rabbitmq:
condition: service_healthy
db:
condition: service_healthy
redis:
condition: service_healthy
networks:
- app_network
restart: unless-stopped
Run Service
taskiq-scheduler:
build:
context: .
dockerfile: src/Dockerfile
command: ["/bin/sh", "-c", "sleep 30 && uv run python -m taskiq scheduler src.messaging.taskiq_broker:scheduler src.messaging.t
environment:
DATABASE_URL: postgresql+asyncpg://user:superpass@db:5432/db
RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
PYTHONPATH: /app
SKIP_MIGRATIONS: "1"
depends_on:
rabbitmq:
condition: service_healthy
db:
condition: service_healthy
redis:
condition: service_healthy
networks:
- app_network
restart: unless-stopped


File Edit Selection View Go Run Terminal Help
EXPLORER
LAB8
pycache
.venv
alembic
src
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
redis_example.py
test_system.py
uv.lock
docker-compose.yml
docker-compose.yml
services:
taskiq-scheduler:
environment:
PYTHONPATH: /app
SKIP_MIGRATIONS: "1"
depends_on:
rabbitmq:
condition: service_healthy
db:
condition: service_healthy
redis:
condition: service_healthy
networks:
- app_network
restart: unless-stopped
volumes:
postgres_data:
rabbitmq_data:
redis-data:
networks:
app_network:

В файл: docker-compose.yml были добавлены 2 сервиса:

- **taskiq-worker**: отвечает за выполнение задач (слушает очередь сообщений, забирает задачи и запускает соответствующие функции)
- **taskiq-scheduler**: отвечает за планирование задач (следит за расписанием и в нужное время отправляет задачи в очередь для выполнения воркерами.

Описание сервиса taskiq-worker:

- sleep 30 - задержка для гарантии запуска зависимостей
- taskiq worker - запуск воркера TaskIQ для выполнения задач
- Указан путь к брокеру: src.messaging.taskiq_broker:taskiq_broker
- Указан модуль с задачами: src.messaging.tasks.report
- restart: unless-stopped - автоматический перезапуск при сбоях

Описание сервиса taskiq-scheduler:

- python -m taskiq scheduler - запуск планировщика TaskIQ
- Указан путь к scheduler: src.messaging.taskiq_broker:scheduler
- Планировщик отслеживает cron-выражения и отправляет задачи в очередь по расписанию
- Воркер забирает задачи из очереди и выполняет их
- Разделение ролей обеспечивает масштабируемость системы

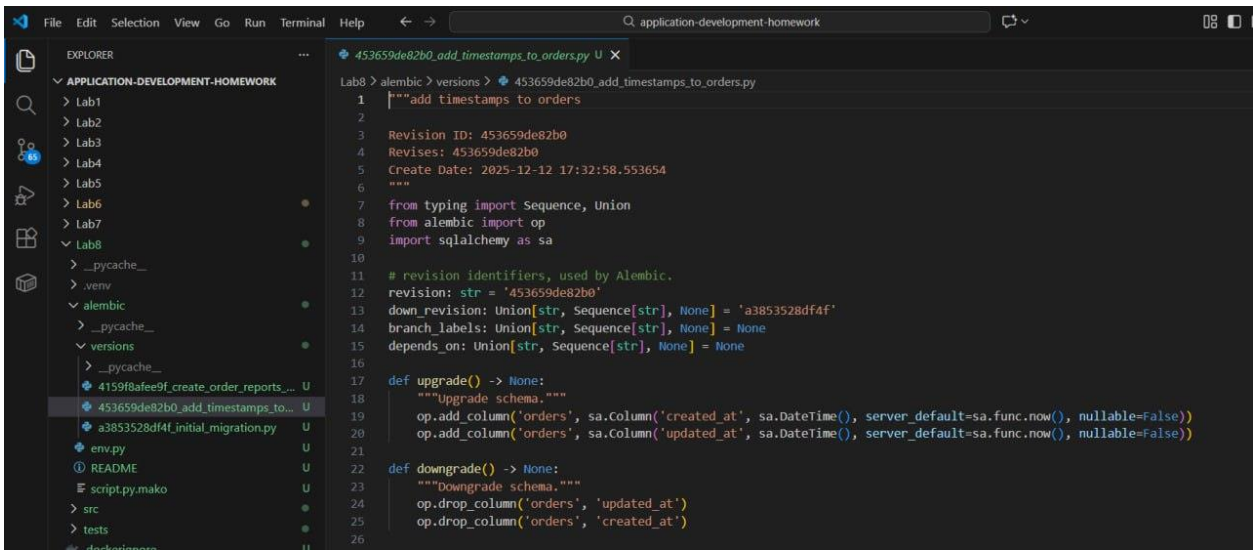
В конфигурации docker-compose.yml для сервисов taskiq-worker и taskiq-scheduler **установлена переменная окружения SKIP_MIGRATIONS: "1",** т.к. миграции уже применены основным приложением при запуске. Это предотвращает конфликты при одновременном доступе к БД.

Часть 1: Создание миграций базы данных

Ход выполнения

Файл: *453659de82b0_add_timestamps_to_orders.py*

(Миграция добавления временных меток к заказам)



Создается view (представление) с нужными полями

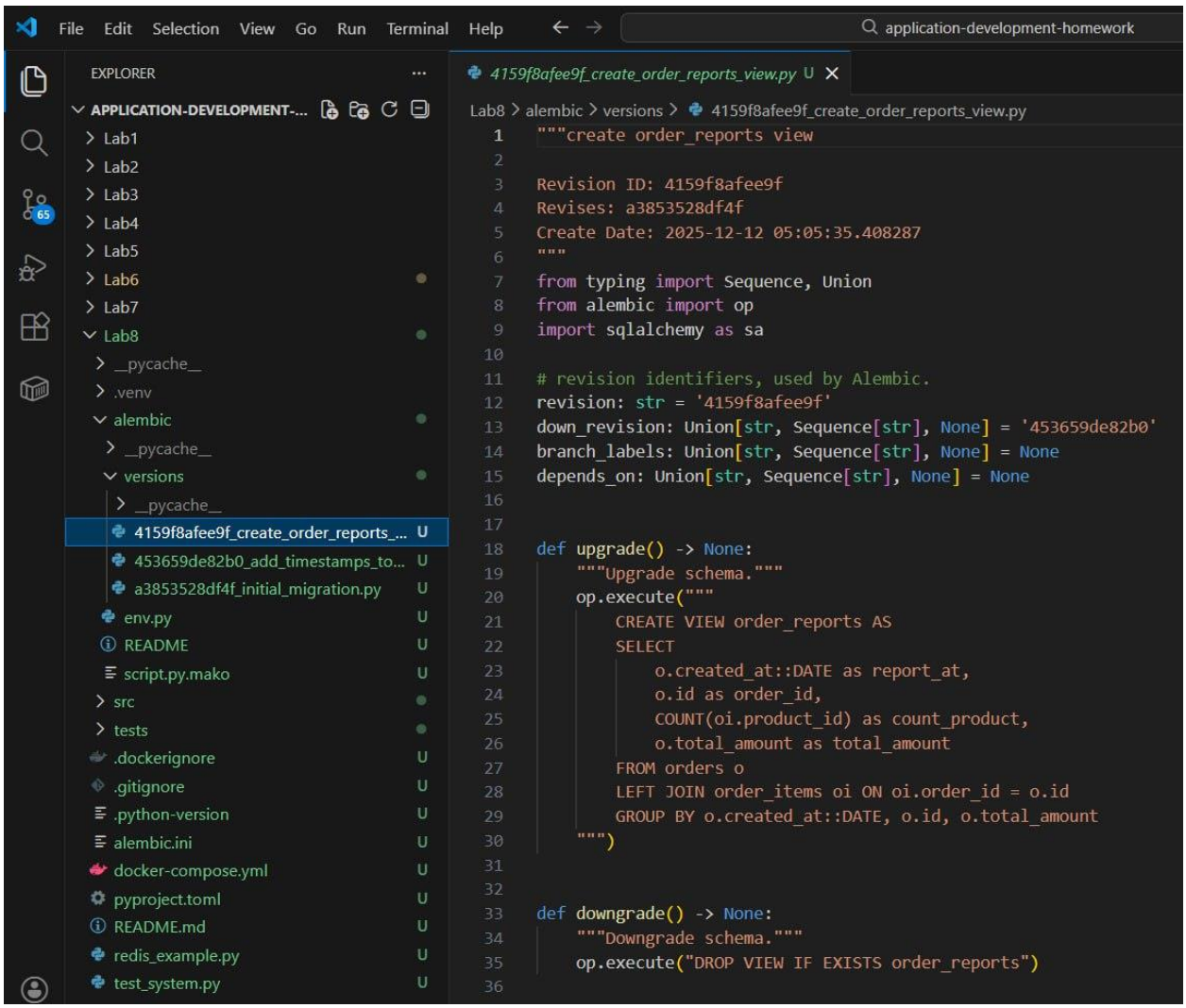
Миграционный файл с именем сгенерирован автоматически с помощью инструмента Alembic после внесения соответствующих изменений в ORM-модель приложения. Имя файла содержит уникальный хеш, обеспечивающий порядок применения миграций.

Цель: расширить таблицу orders полями created_at и updated_at для возможности корректной группировки и анализа заказов по датам, что является обязательным для формирования отчетов.

- поля имеют тип DateTime и автоматически заполняются текущим временем при создании записи
- server_default=sa.func.now() обеспечивает установку времени на уровне базы данных в момент вставки или обновления записи, что исключает ошибки из-за разницы во времени на сервере приложения и сервере БД
- nullable=False делает поля обязательными, обеспечивая целостность данных для всех новых записей.

Файл: *4159f8afee9f_create_order_reports_view.py*

(Миграция создания представления для отчетов)



Создается view (представление) с нужными полями

Цель: создать виртуальную таблицу order_reports, которая предоставляет уже сгруппированные и готовые к использованию данные для отчетов, избавляя прикладной код от необходимости выполнения сложных JOIN-запросов каждый раз.

Описание:

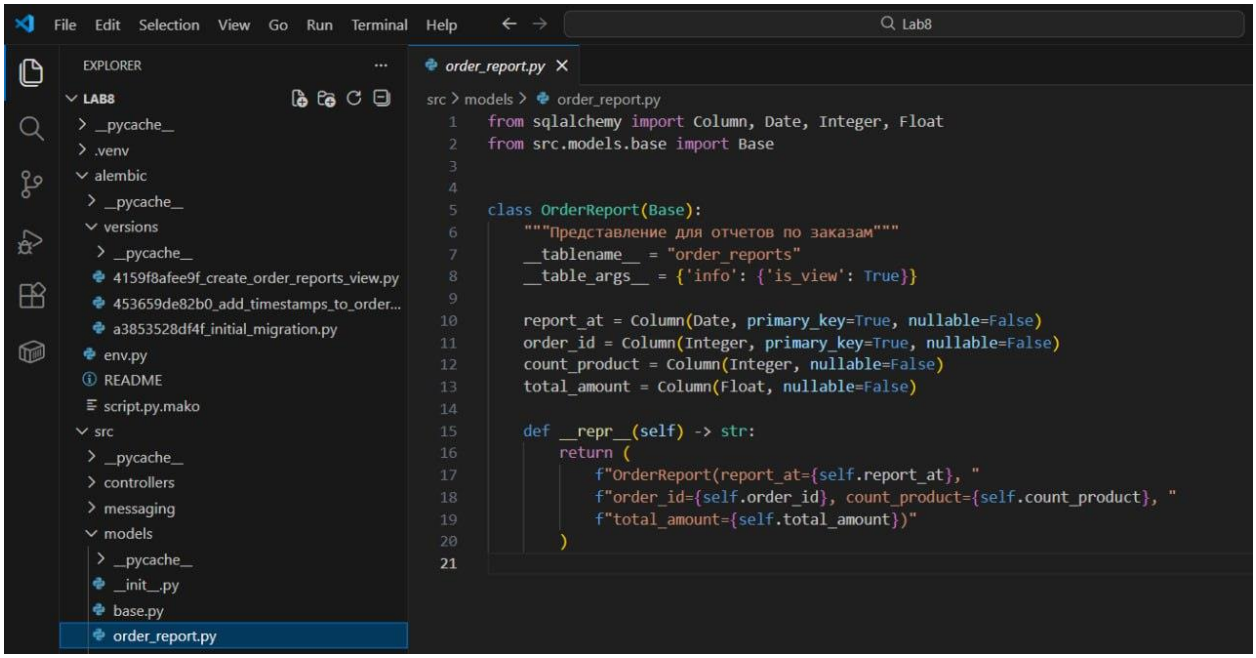
- представление order_reports - объединяет данные из таблиц orders и order_items
- o.created_at::DATE - преобразование timestamp в дату для группировки по дням
- COUNT(oi.product_id) - подсчет количества товаров в каждом заказе
- LEFT JOIN - гарантирует включение заказов даже без товаров

- GROUP BY - группирует данные по дате создания заказа, ID заказа и общей сумме

Часть 2: Создание модели OrderReport

Файл: *order_report.py*

(**Определена ORM-модель для работы с представлением SQLAlchemy**)



- Создана SQLAlchemy модель для работы с представлением order_reports
- __table_args__ = {'info': {'is_view': True}} - указывает, что это VIEW, а не таблица
- Композитный первичный ключ из report_at и order_id обеспечивает уникальность записей
- Модель содержит все необходимые поля для отчета: дата, ID заказа, количество продуктов и общая сумма

Порядок работы:

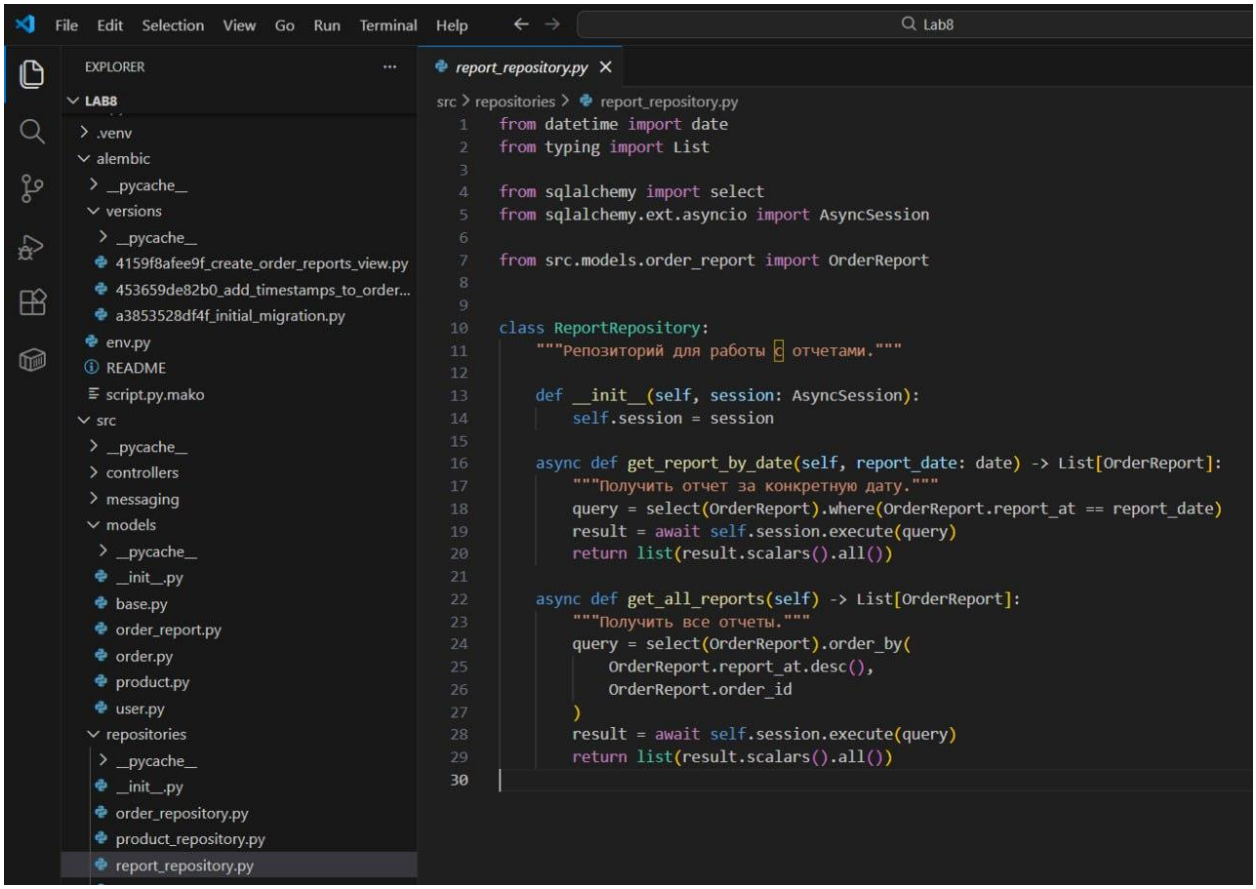
1) Миграция создает VIEW в PostgreSQL - при выполнении alembic upgrade head создается SQL-представление, которое автоматически

агрегирует данные из таблиц orders и order_items.

2) Модель позволяет работать с VIEW как с обычной таблицей, можно делать SELECT-запросы через SQLAlchemy ORM, и данные будут автоматически браться из представления.
3) VIEW автоматически обновляется, каждый раз при SELECT-запросе PostgreSQL выполняет заложенный в VIEW запрос и возвращает актуальные данные. Т.е. не нужно вручную обновлять отчеты.

Часть 3: Создание репозитория для отчетов

Файл: *report_repository.py*



Репозиторий является ключевым связующим звеном между созданным SQL - представлением и бизнес-логикой приложения (задачами планировщика и REST API).

Описание методов:

- get_report_by_date() - получает отчеты за конкретную дату с

фильтрацией по report_at

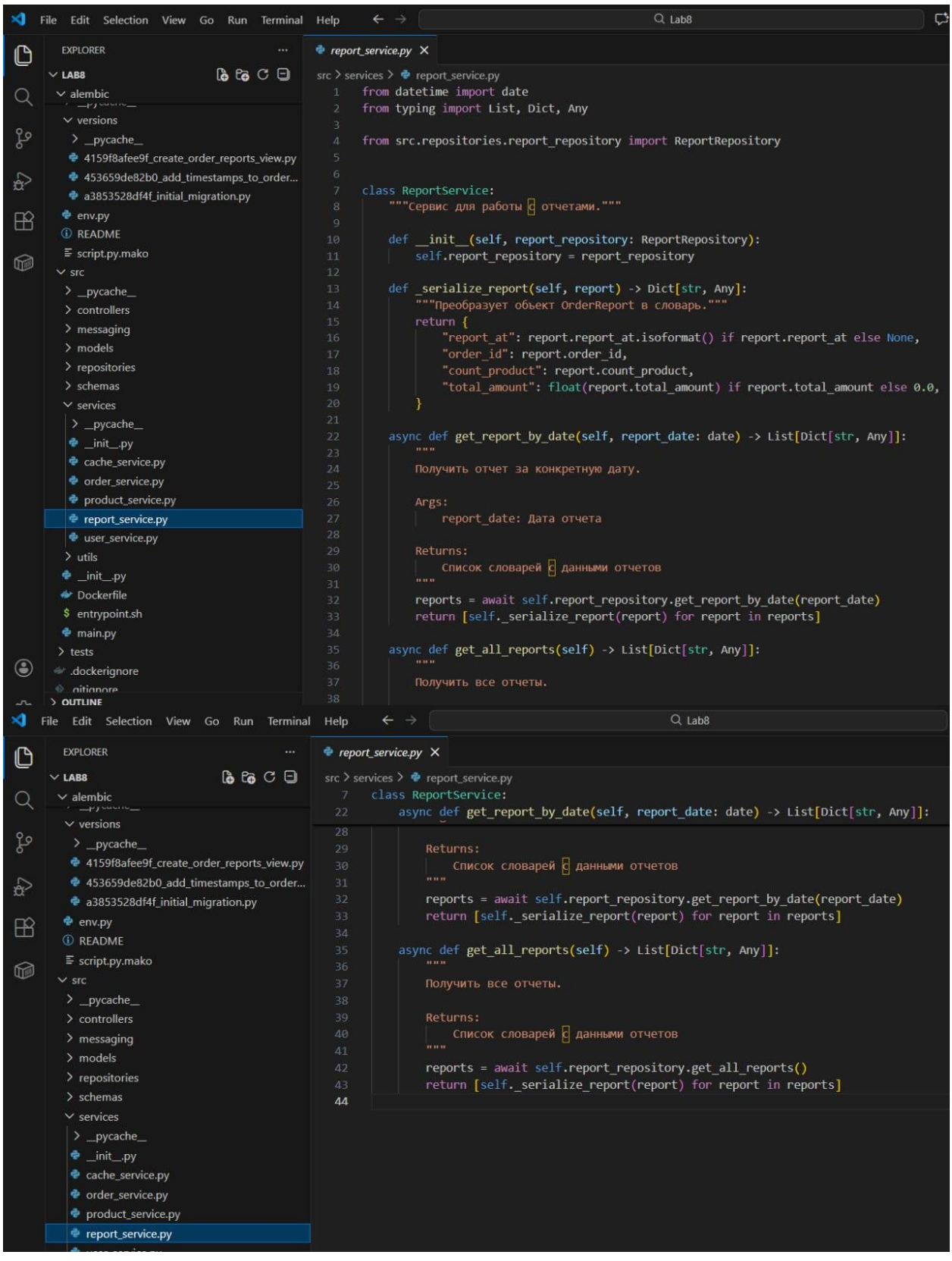
- select(OrderReport) — использует модель представления.
- .where(OrderReport.report_at == report_date) - фильтрация по дате. Поле report_at имеет тип DATE благодаря преобразованию DATE в SQL-представлении.

- get_all_reports() - получает все отчеты с сортировкой по дате (от новых к старым) и ID заказа
  - .order_by(OrderReport.report_at.desc(), OrderReport.order_id) - сортировка сначала по дате отчета (новые сверху), затем по ID заказа для детерминированности.

Репозиторий построен на асинхронной сессии SQLAlchemy (AsyncSession), что важно для работы в асинхронном окружении FastAPI и TaskIQ без блокировки операций ввода-вывода

Часть 4: Создание сервиса для работы с отчетами

Файл: *report_service.py*



Цель создания ReportService: вынести бизнес-логику преобразования данных из контроллеров API и задач планировщика в отдельный, переиспользуемый компонент. Его задача - не доставать данные (это делает репозиторий), а готовить их для конечного потребителя. Это слой бизнес-логики между

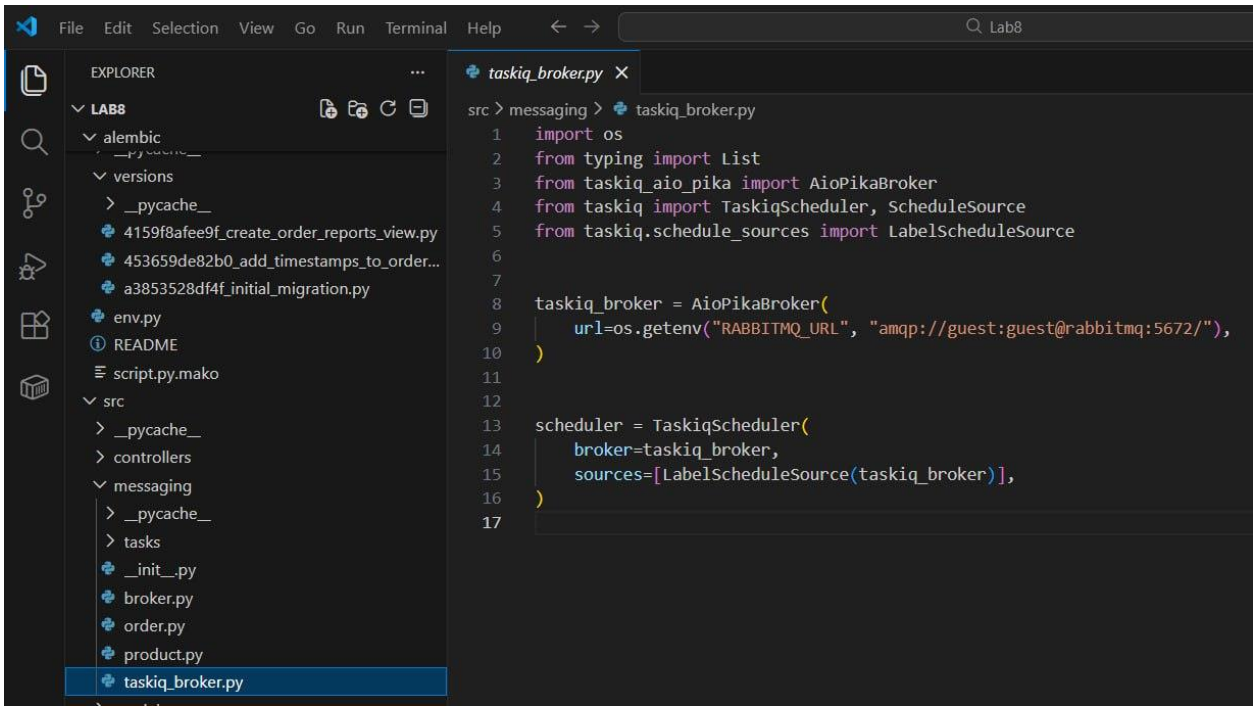
контроллером и репозиторием.

Описание работы методов:

- _serialize_report() - приватный метод для преобразования ORM-объектов в JSON-сериализуемые словари. Метод конвертирует даты в ISO формат строки для корректной JSON-сериализации и обеспечивает безопасное преобразование типов (float для total_amount)
- get_report_by_date() – метод предоставляет единый интерфейс для запроса отчета за дату и передает в ответ готовый к использованию список словарей.
- get_all_reports() – позволяет получить все записи из представления order_reports

Часть 5: Создание планировщика

Файл: *taskiq_broker.py*



Ход выполнения:

Цель: сформировать инфраструктуру для декларативного создания, расписания и выполнения фоновых задач. Вынос этой конфигурации в отдельный модуль (taskiq_broker.py) необходим для обеспечения чистоты

архитектуры и повторного использования.

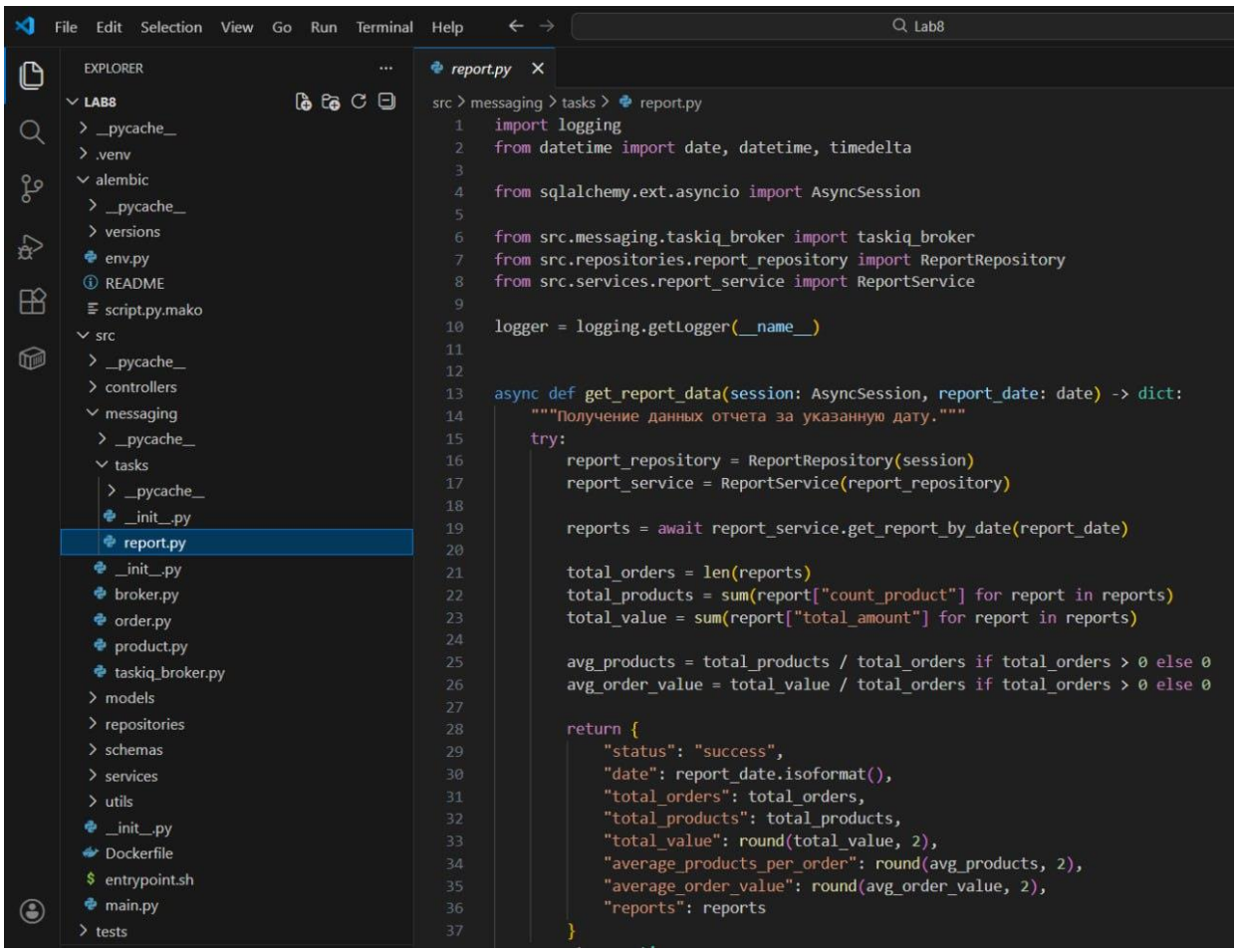
Описание работы:

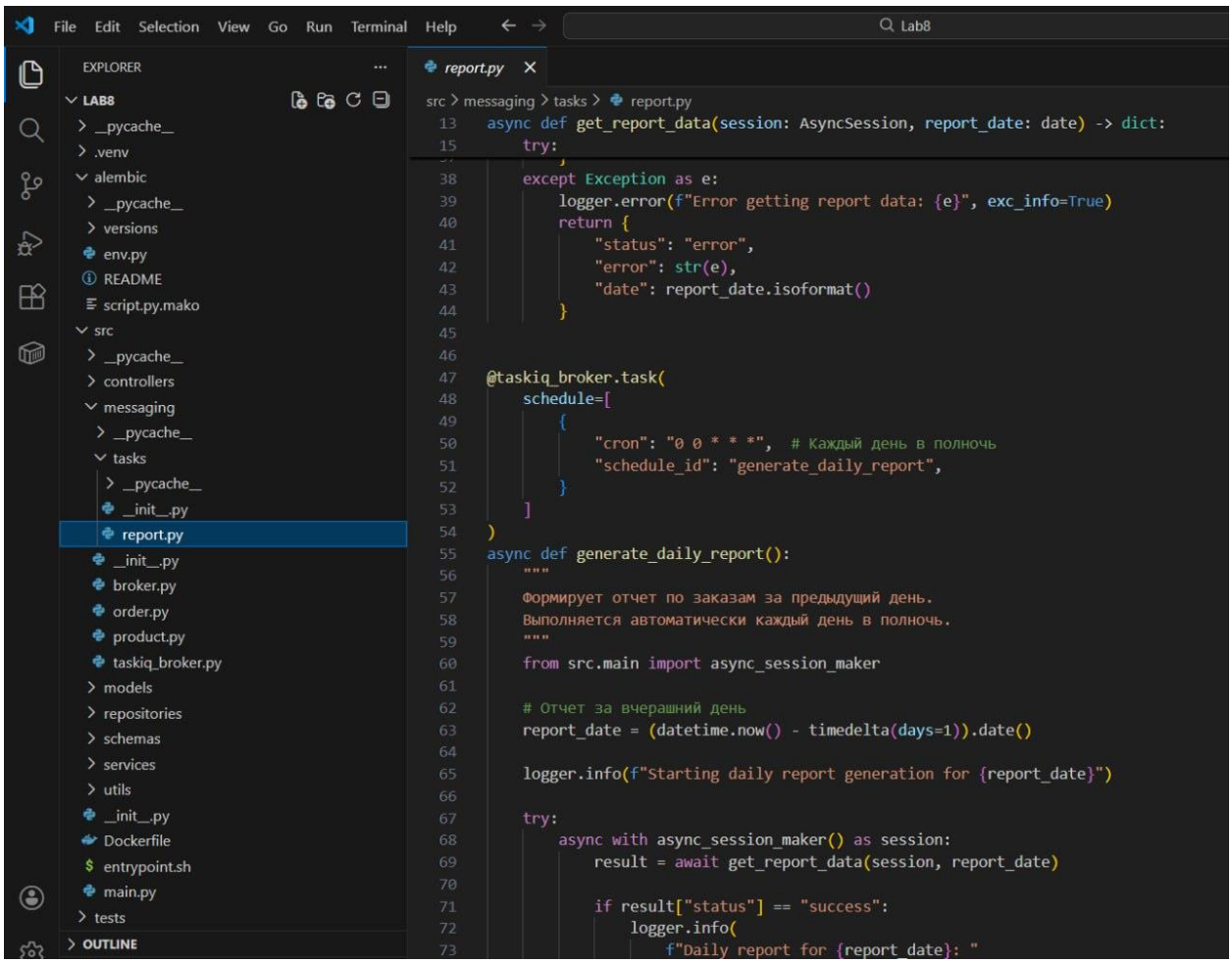
- taskiq_broker - брокер сообщений на базе RabbitMQ для асинхронного выполнения задач
- AioPikaBroker использует asyncio-совместимый клиент для RabbitMQ
- Создан scheduler - планировщик для выполнения задач по расписанию
- LabelScheduleSource позволяет определять расписание через декораторы задач с помощью параметра schedule.

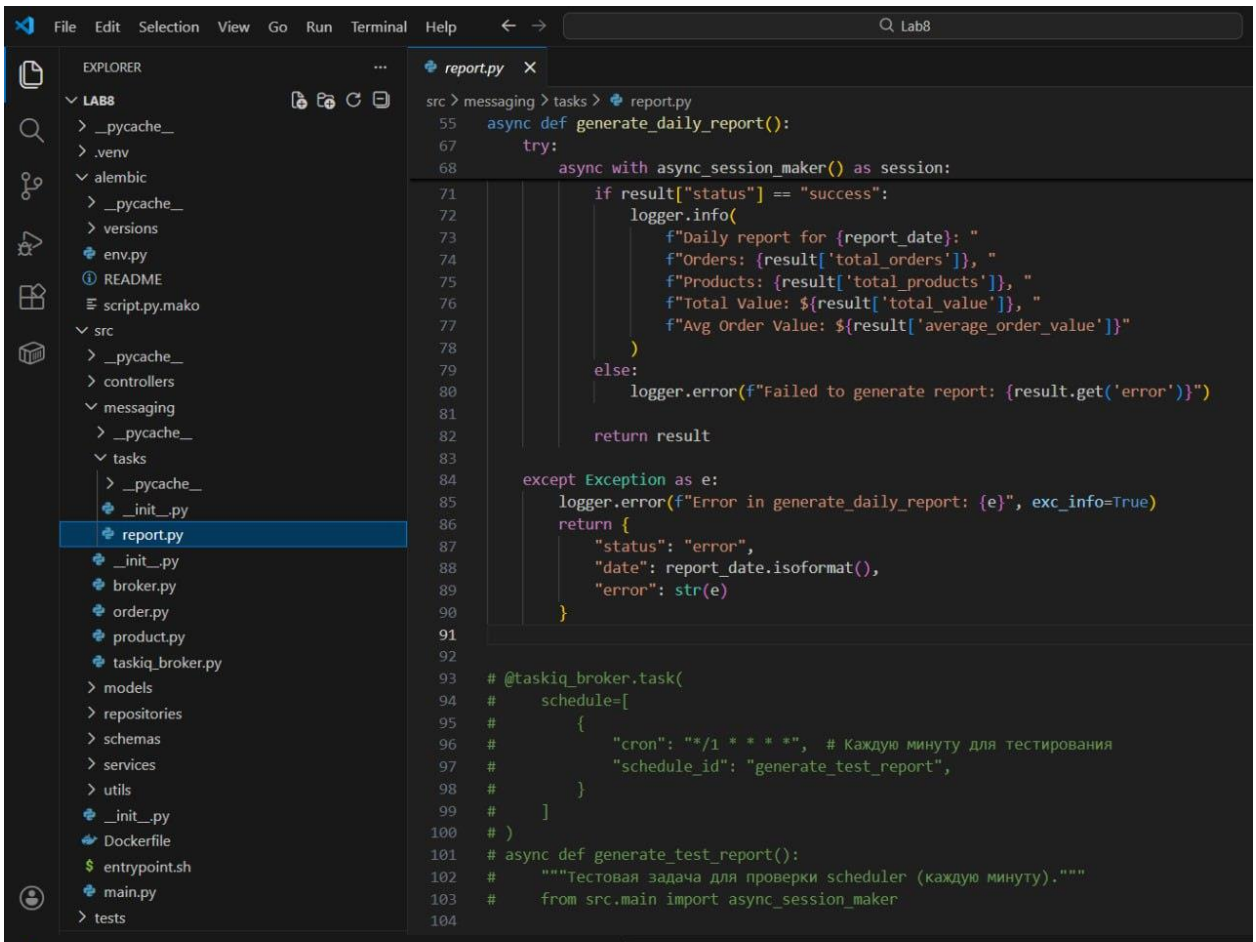
Запуск компонентов (через Docker Compose):

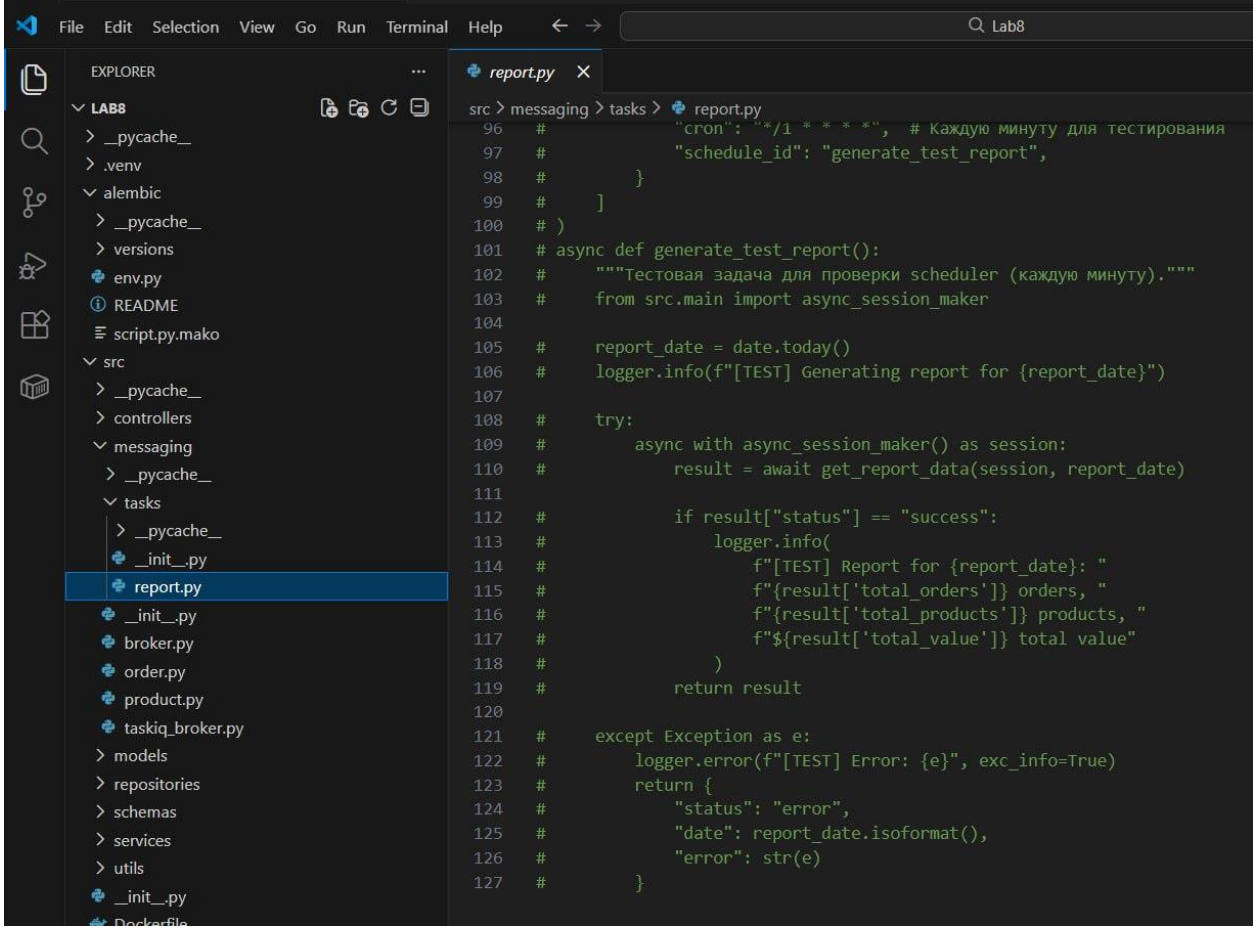
- Сервис taskiq-scheduler в docker-compose.yml запускает этот планировщик (python -m taskiq scheduler ...).
- Сервис taskiq-worker запускает воркеров, которые подписываются на taskiq_broker для получения и выполнения задач.

Файл*: report.py*
(Реализация задачи генерации отчетов)


File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
report.py
src > messaging > tasks > report.py
import logging
from datetime import date, datetime, timedelta
from sqlalchemy.ext.asyncio import AsyncSession
from src.messaging.taskiq_broker import taskiq_broker
from src.repositories.report_repository import ReportRepository
from src.services.report_service import ReportService
logger = logging.getLogger(__name__)
async def get_report_data(session: AsyncSession, report_date: date) -> dict:
"""Получение данных отчета за указанную дату."""
try:
report_repository = ReportRepository(session)
report_service = ReportService(report_repository)
reports = await report_service.get_report_by_date(report_date)
total_orders = len(reports)
total_products = sum(report["count_product"] for report in reports)
total_value = sum(report["total_amount"] for report in reports)
avg_products = total_products / total_orders if total_orders > 0 else 0
avg_order_value = total_value / total_orders if total_orders > 0 else 0
return {
"status": "success",
"date": report_date.isoformat(),
"total_orders": total_orders,
"total_products": total_products,
"total_value": round(total_value, 2),
"average_products_per_order": round(avg_products, 2),
"average_order_value": round(avg_order_value, 2),
"reports": reports
}


File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
report.py
src > messaging > tasks > report.py
async def get_report_data(session: AsyncSession, report_date: date) -> dict:
try:
except Exception as e:
logger.error(f"Error getting report data: {e}", exc_info=True)
return {
"status": "error",
"error": str(e),
"date": report_date.isoformat()
}
@taskiq_broker.task(
schedule=[
{
"cron": "0 0 * * *", # Каждый день в полночь
"schedule_id": "generate_daily_report",
}
]
)
async def generate_daily_report():
"""
Формирует отчет по заказам за предыдущий день.
Выполняется автоматически каждый день в полночь.
"""
from src.main import async_session_maker
# Отчет за вчерашний день
report_date = (datetime.now() - timedelta(days=1)).date()
logger.info(f"Starting daily report generation for {report_date}")
try:
async with async_session_maker() as session:
result = await get_report_data(session, report_date)
if result["status"] == "success":
logger.info(
f"Daily report for {report_date}: "
OUTLINE


File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
report.py
src > messaging > tasks > report.py
async def generate_daily_report():
try:
async with async_session_maker() as session:
if result["status"] == "success":
logger.info(
f"Daily report for {report_date}: "
f"Orders: {result['total_orders']}, "
f"Products: {result['total_products']}, "
f"Total Value: ${result['total_value']}, "
f"Avg Order Value: ${result['average_order_value']}"
)
else:
logger.error(f"Failed to generate report: {result.get('error')}")
return result
except Exception as e:
logger.error(f"Error in generate_daily_report: {e}", exc_info=True)
return {
"status": "error",
"date": report_date.isoformat(),
"error": str(e)
}
# @taskiq_broker.task(
# schedule=[
# {
# "cron": "*/1 * * * *", # Каждую минуту для тестирования
# "schedule_id": "generate_test_report",
# }
#]
#)
async def generate_test_report():
"""Тестовая задача для проверки scheduler (каждую минуту)."""
from src.main import async_session_maker
pycache
.venv
alembic
pycache
versions
env.py
README
script.py.mako
src
pycache
controllers
messaging
pycache
tasks
pycache
init.py
report.py
init.py
broker.py
order.py
product.py
taskiq_broker.py
models
repositories
schemas
services
utils
init.py
Dockerfile
entrypoint.sh
main.py
tests


File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
report.py
src > messaging > tasks > report.py
"cron": "*/1 * * * *", # Каждую минуту для тестирования
"schedule_id": "generate_test_report",
}
]
)
async def generate_test_report():
"""Тестовая задача для проверки scheduler (каждую минуту)."""
from src.main import async_session_maker
report_date = date.today()
logger.info(f"[TEST] Generating report for {report_date}")
try:
async with async_session_maker() as session:
result = await get_report_data(session, report_date)
if result["status"] == "success":
logger.info(
f"[TEST] Report for {report_date}: "
f"{result['total_orders']} orders, "
f"{result['total_products']} products, "
f"${result['total_value']} total value"
)
return result
except Exception as e:
logger.error(f"[TEST] Error: {e}", exc_info=True)
return {
"status": "error",
"date": report_date.isoformat(),
"error": str(e)
}
pycache
.venv
alembic
pycache
versions
env.py
README
script.py.mako
src
pycache
controllers
messaging
pycache
tasks
pycache
init.py
report.py
init.py
broker.py
order.py
product.py
taskiq_broker.py
models
repositories
schemas
services
utils
init.py

Функция get_report_data (session, report_date) - использует сервисный слой для получения данных, а затем применяет к ним дополнительную бизнес-логику - вычисление сводной статистики

- Инициализация цепочки: Создаёт экземпляры ReportRepository и ReportService. Это демонстрирует правильное использование Dependency Injection даже внутри задачи.
- Получение данных: Через report_service.get_report_by_date(report_date) получает уже сериализованные данные из представления.
- Агрегация: Рассчитывает ключевые метрики (total_orders, total_value и т.д.), превращая список отдельных записей в стратегическую сводку для менеджмента.
- Обработка краевых случаев: Проверка if total_orders > 0 предотвращает деление на ноль, что критически важно для отказоустойчивости (например, если в выходной день заказов не было).
- Структурированный ответ и обработка ошибок: Функция всегда возвращает словарь с полем "status", что позволяет вызывающему коду легко понять результат. Все исключения перехватываются, логируются с полным traceback (exc_info=True) и возвращаются в структурированном виде, а не "выбрасываются" наружу, что могло бы привести к падению всего воркера.

Задача generate_daily_report() - точка входа, управляемая планировщиком.

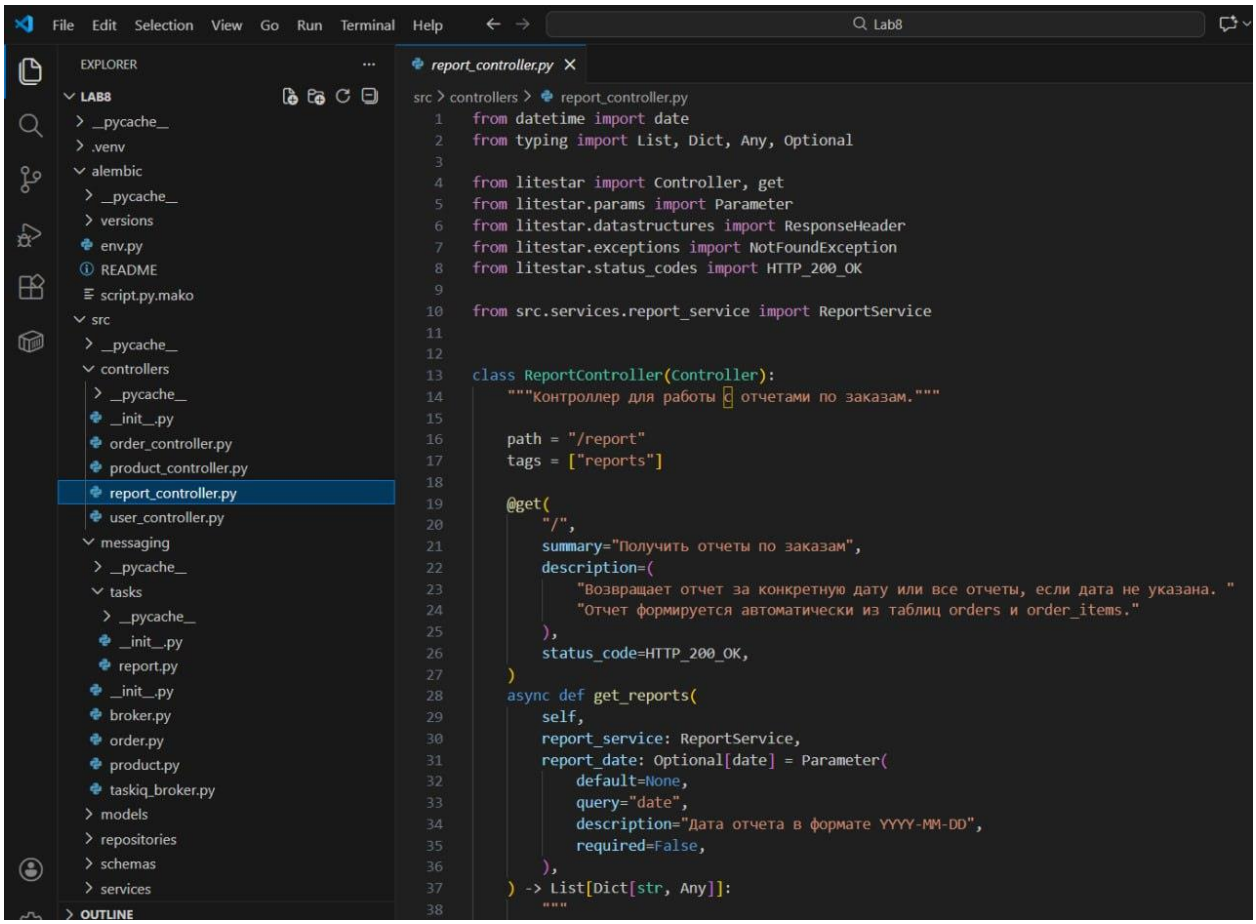
- Декоратор @taskiq_broker.task регистрирует функцию как задачу TaskIQ
- schedule определяет расписание выполнения:
  - "cron": "0 0 * * *" — cron-выражение (каждый день в 00:00)
  - "schedule_id" — уникальный идентификатор задачи для отслеживания
- Задача формирует отчет за вчерашний день (datetime.now() - timedelta(days=1))

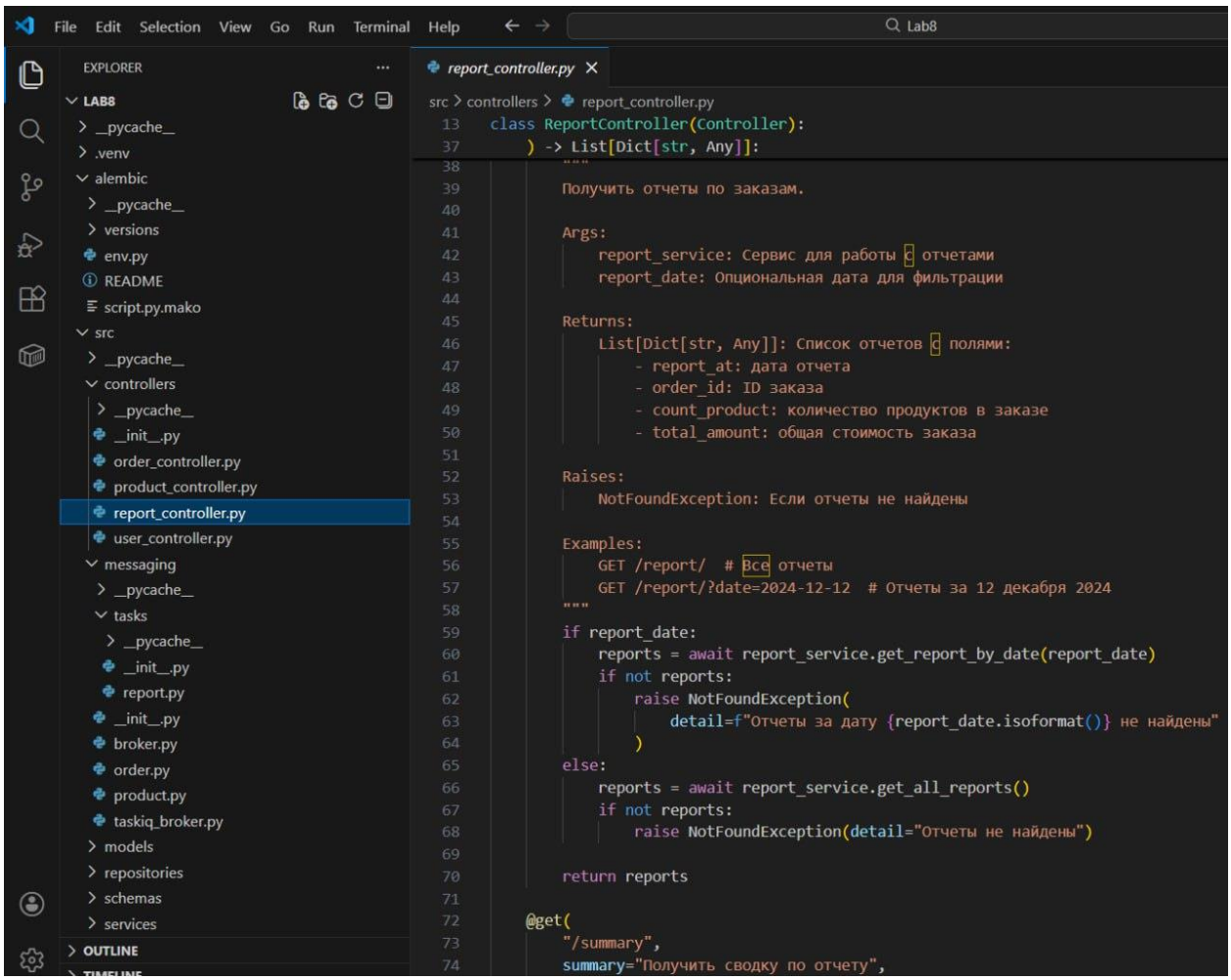
- Использует контекстный менеджер для автоматического управления сессией БД
- Логирует начало выполнения, результаты и ошибки
- Возвращает структурированный результат для мониторинга

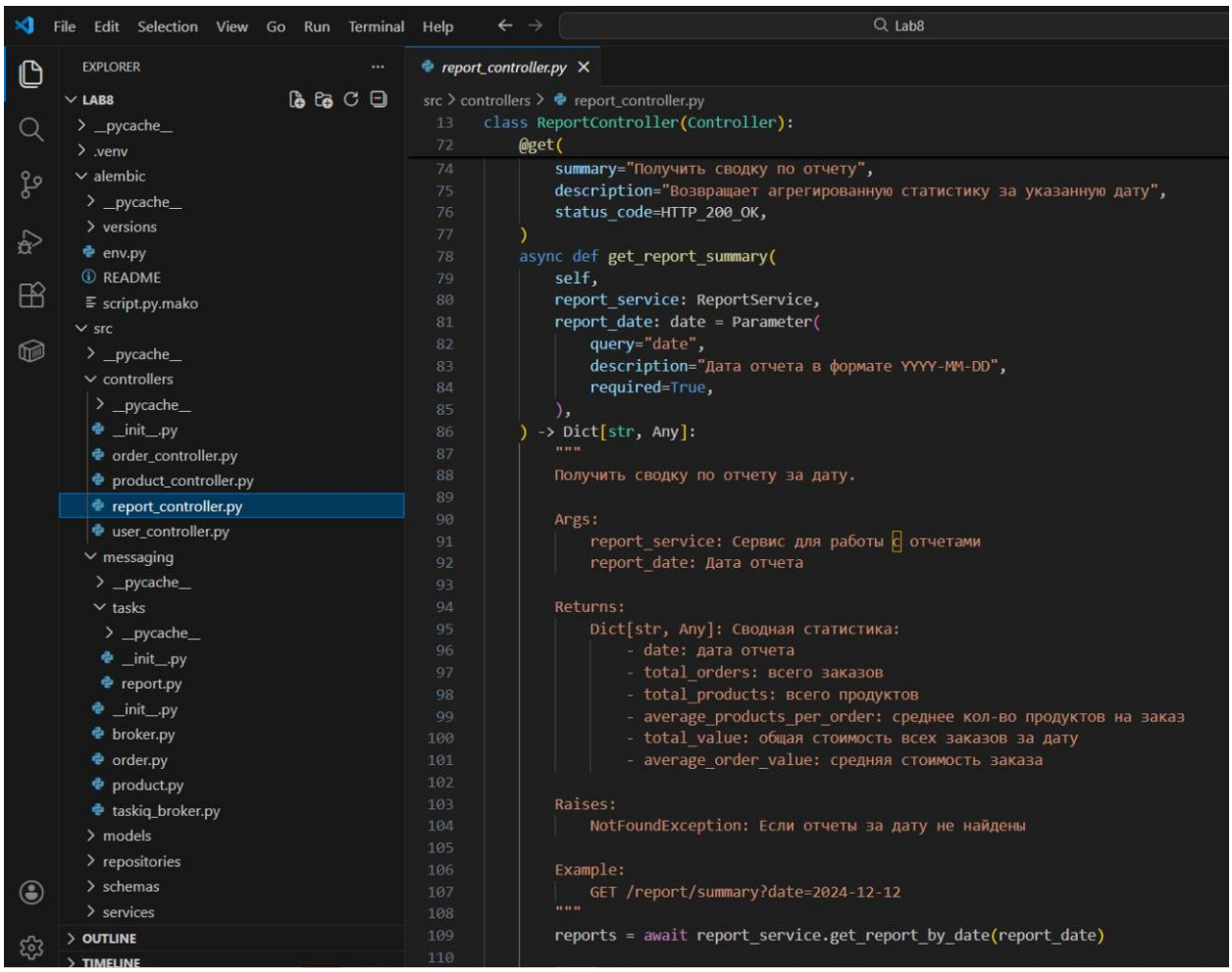
Часть 6: Создание отчета по заказам

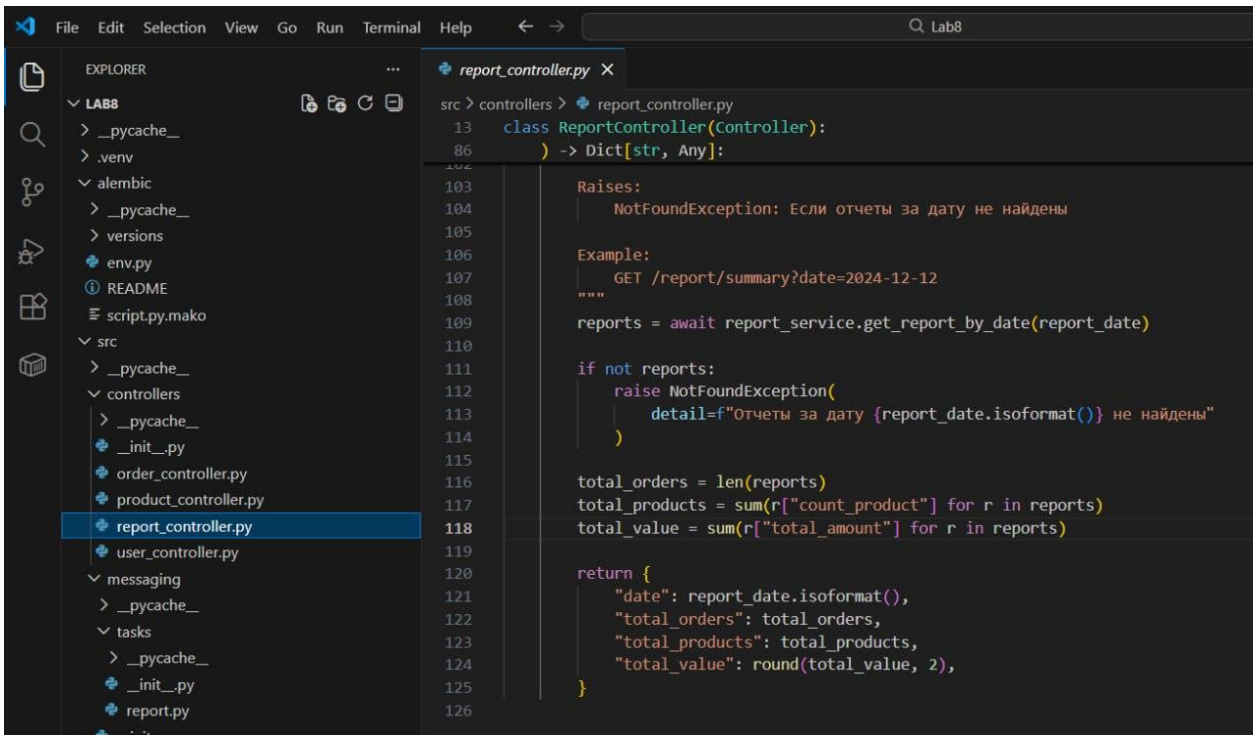
Файл: *report_controller.py*

(Создание REST API контроллера)




File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
pycache
.venv
alembic
pycache
versions
env.py
README
script.py.mako
src
pycache
controllers
pycache
__init__.py
order_controller.py
product_controller.py
report_controller.py
user_controller.py
messaging
pycache
tasks
pycache
__init__.py
report.py
__init__.py
broker.py
order.py
product.py
taskiq_broker.py
models
repositories
schemas
services
OUTLINE
report_controller.py
src > controllers > report_controller.py
class ReportController(Controller):
) -> List[Dict[str, Any]]:
"""
Получить отчеты по заказам.
Args:
report_service: Сервис для работы с отчетами
report_date: Опциональная дата для фильтрации
Returns:
List[Dict[str, Any]]: Список отчетов с полями:
- report_at: дата отчета
- order_id: ID заказа
- count_product: количество продуктов в заказе
- total_amount: общая стоимость заказа
Raises:
NotFoundException: Если отчеты не найдены
Examples:
GET /report/ # Все отчеты
GET /report/?date=2024-12-12 # Отчеты за 12 декабря 2024
"""
if report_date:
reports = await report_service.get_report_by_date(report_date)
if not reports:
raise NotFoundException(
detail=f"Отчеты за дату {report_date.isoformat()} не найдены"
)
else:
reports = await report_service.get_all_reports()
if not reports:
raise NotFoundException(detail="Отчеты не найдены")
return reports
@get(
"/summary",
summary="Получить сводку по отчету",





Ход выполнения:

Цель: Создать HTTP-интерфейс для ручного доступа к данным отчетов,

дополняющий автоматическую генерацию. Контроллер предоставляет 2 эндпоинта, которые используют ранее созданный ReportService.

Ключевые эндпоинты:

1) **GET /report/**
   - возвращает либо все отчеты, либо отчет за конкретную дату
   - параметр date в запросе не обязателен, если он указан - контроллер вызывает report_service.get_report_by_date(), а если нет - report_service.get_all_reports()
   - инжекция зависимости report_service через Litestar DI
   - при отсутствии данных возвращается ошибка NotFoundException (статус 404), а не пустой список

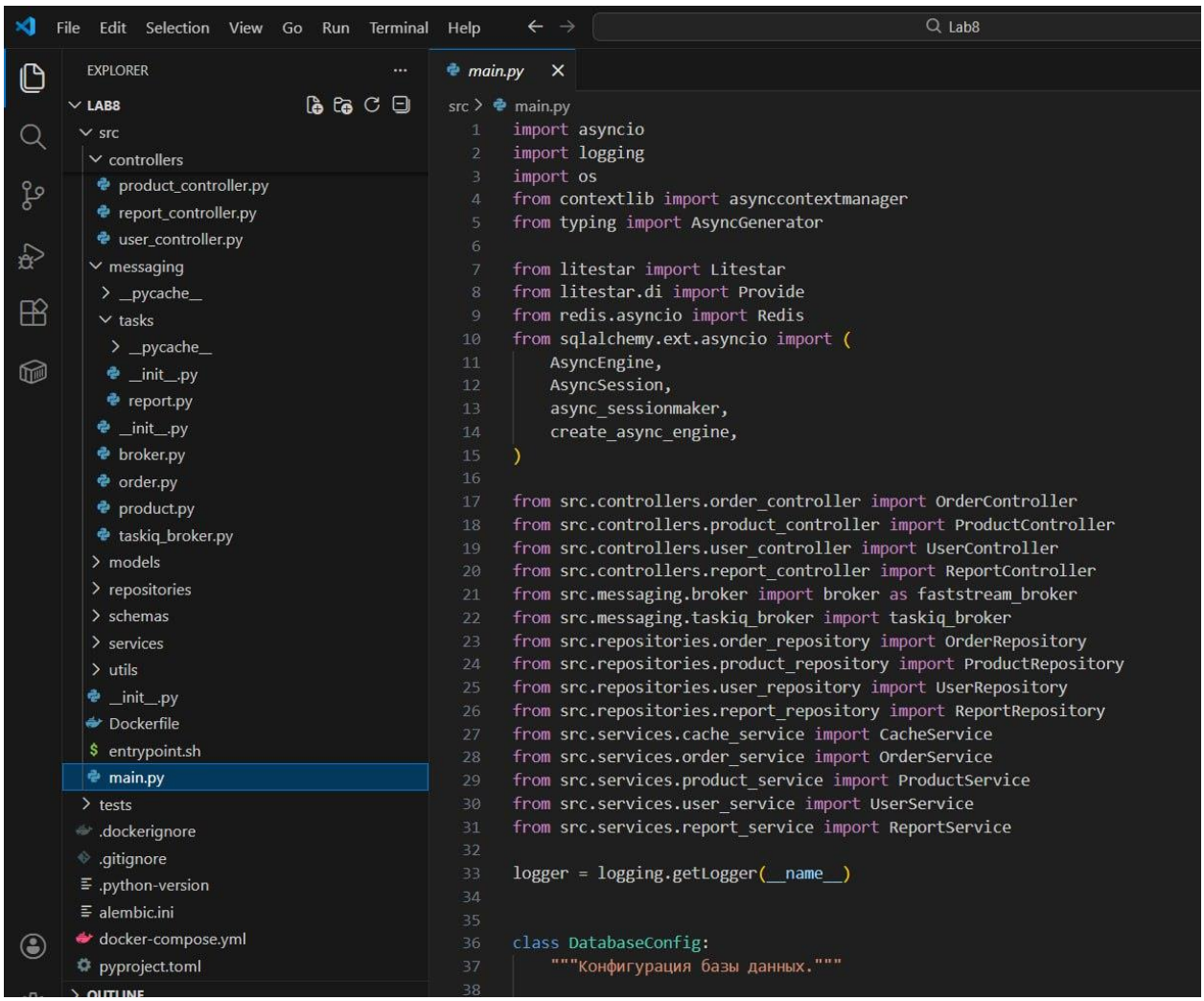
2) **GET /report/summary**
   - возвращает не детальный список, а сводную агрегированную статистику за указанный день.
   - обязательный параметр date (возвращает сводную информацию: дата отчета, общее количество заказов, общее количество продуктов, общая стоимость всех заказов)
   - контроллер получает детали через сервис, а затем сам вычисляет общие суммы и количество прямо в коде

Файл: *main.py*

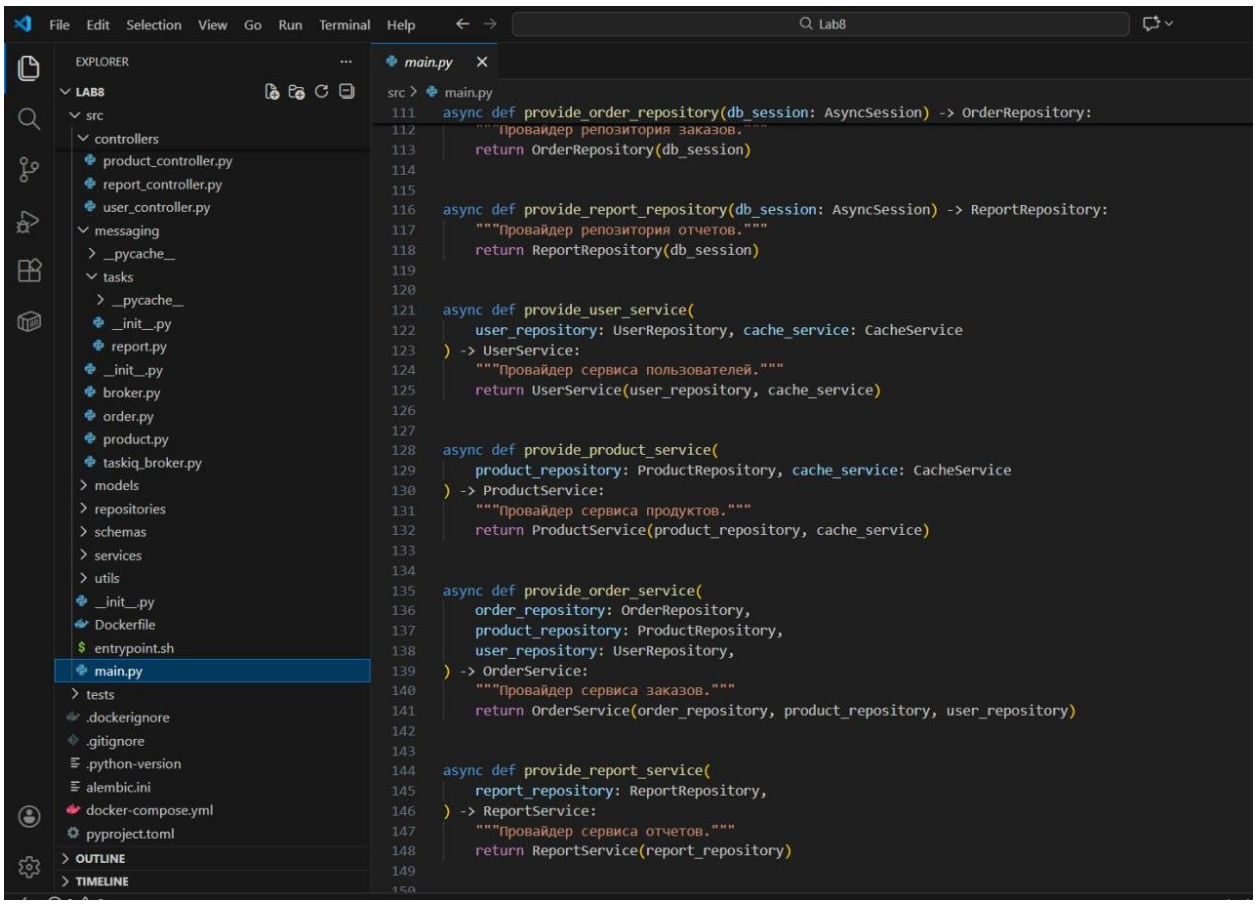
(Интеграция в главный модуль приложения)

Ход выполнения:

Цель: собрать все созданные модули (контроллер, сервис, задачи) в единое приложение, настроив систему зависимостей и запуска.



1. В main.py были добавлены импорты ReportController, а также ReportRepository и ReportService.

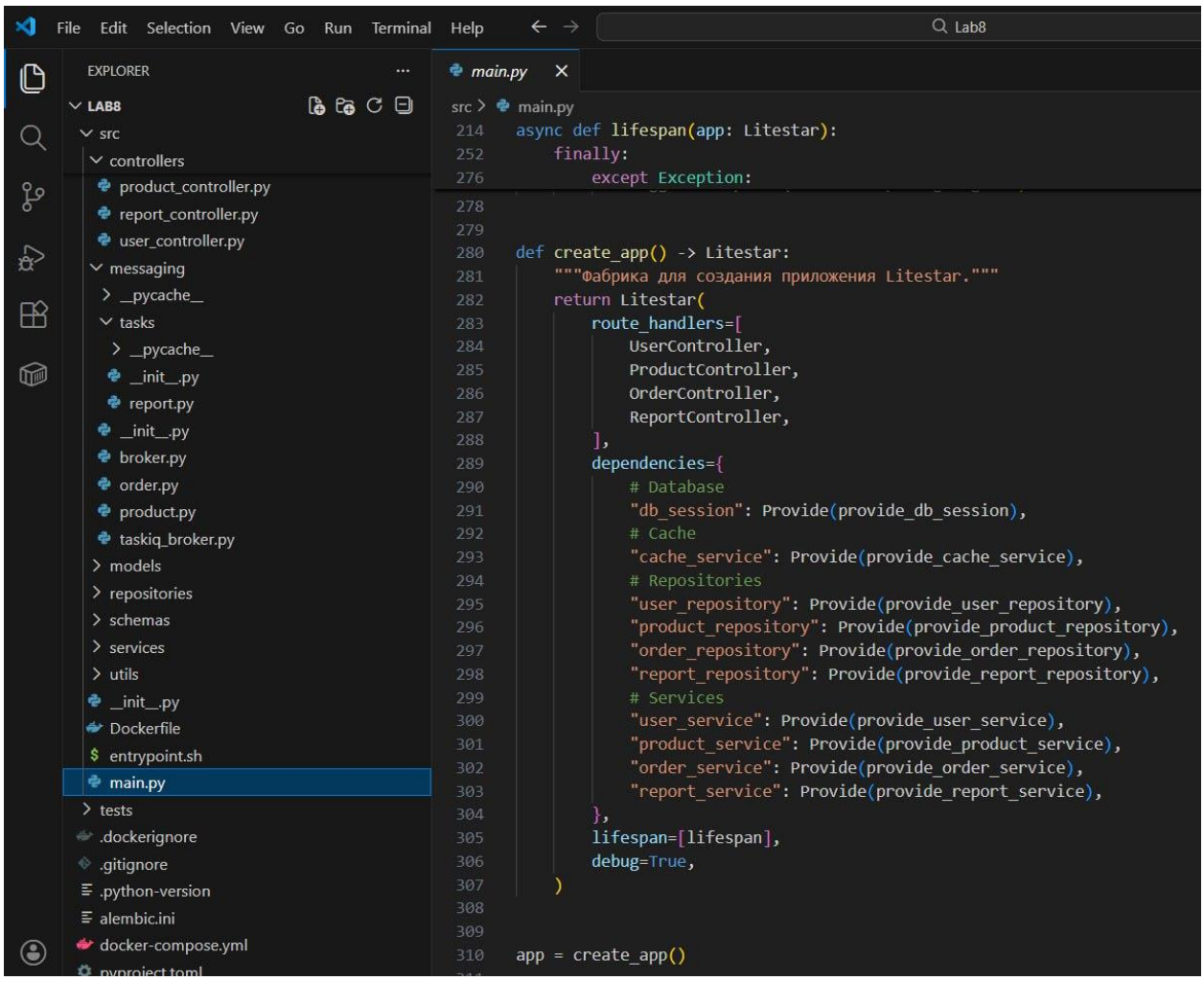


2. Добавлены провайдеры зависимостей:

- provide_report_repository() – провайдер репозитория отчетов
- provide_report_service() - провайдер сервиса отчетов

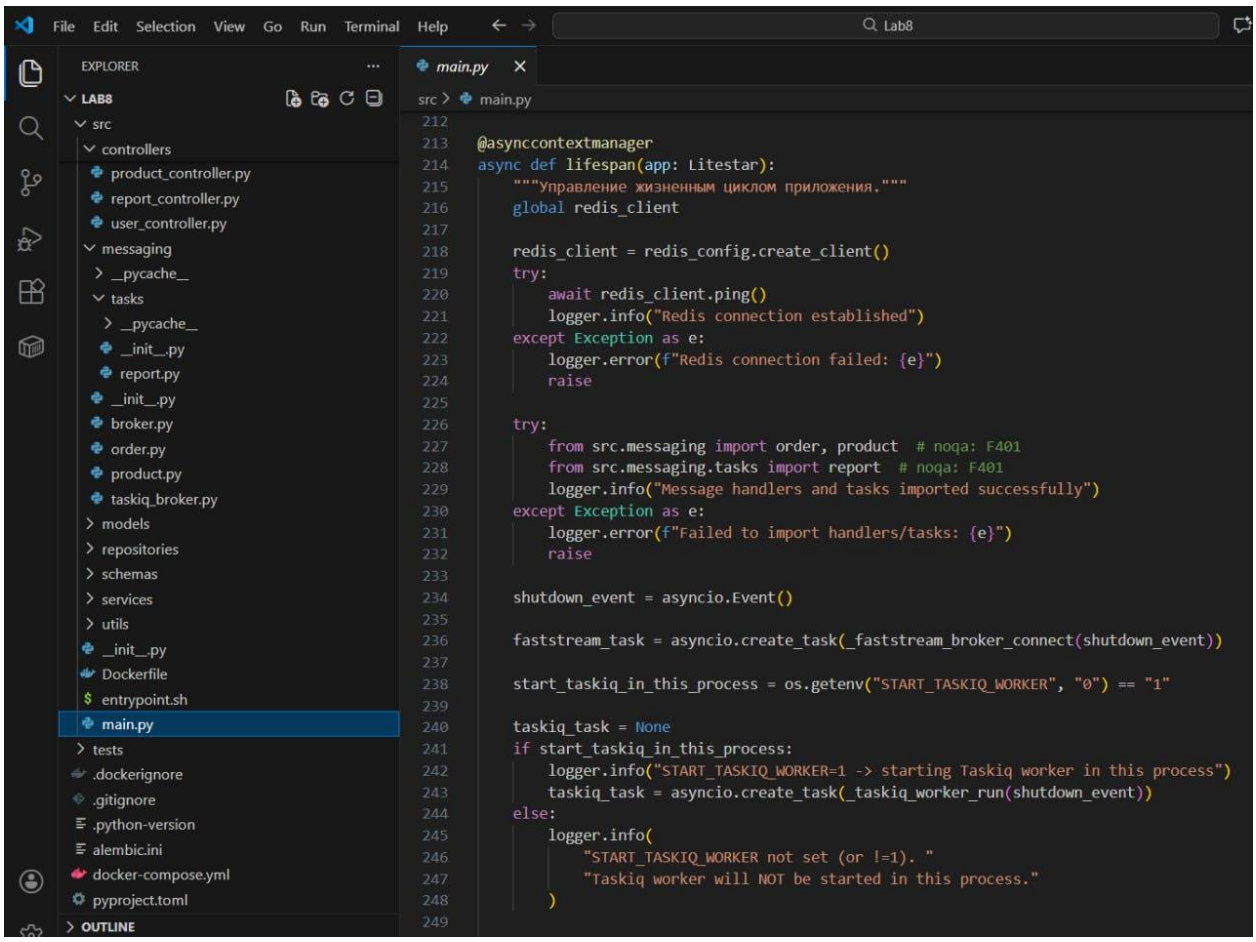
Описание:

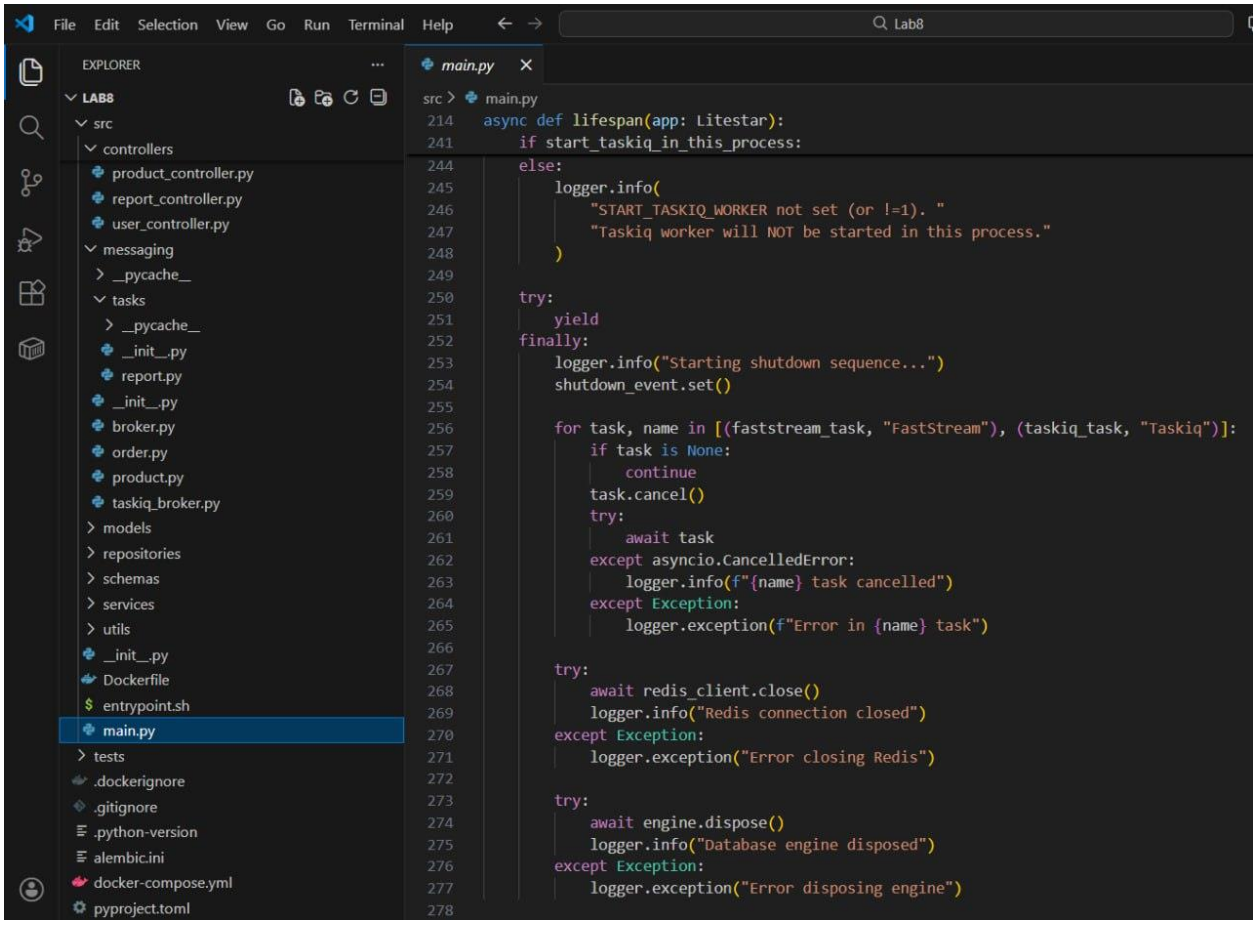
- Функции-провайдеры создают экземпляры классов с правильными зависимостями
- Litestar автоматически инжектит эти зависимости в контроллеры
- Обеспечивает единый жизненный цикл объектов в рамках запроса



3. Регистрация в приложении:

- ReportController добавлен в список route_handlers - так приложение узнаёт о новых REST API эндпоинтах (/report).
- Провайдеры report_repository и report_service зарегистрированы в общем словаре dependencies для глобального использования.


File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
src
controllers
product_controller.py
report_controller.py
user_controller.py
messaging
__pycache__
tasks
__pycache__
__init__.py
report.py
__init__.py
broker.py
order.py
product.py
taskiq_broker.py
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
OUTLINE
main.py
src > main.py
@asynccontextmanager
async def lifespan(app: Litestar):
"""Управление жизненным циклом приложения."""
global redis_client
redis_client = redis_config.create_client()
try:
await redis_client.ping()
logger.info("Redis connection established")
except Exception as e:
logger.error(f"Redis connection failed: {e}")
raise
try:
from src.messaging import order, product # noqa: F401
from src.messaging.tasks import report # noqa: F401
logger.info("Message handlers and tasks imported successfully")
except Exception as e:
logger.error(f"Failed to import handlers/tasks: {e}")
raise
shutdown_event = asyncio.Event()
faststream_task = asyncio.create_task(_faststream_broker_connect(shutdown_event))
start_taskiq_in_this_process = os.getenv("START_TASKIQ_WORKER", "0") == "1"
taskiq_task = None
if start_taskiq_in_this_process:
logger.info("START_TASKIQ_WORKER=1 -> starting Taskiq worker in this process")
taskiq_task = asyncio.create_task(_taskiq_worker_run(shutdown_event))
else:
logger.info(
"START_TASKIQ_WORKER not set (or !=1). "
"Taskiq worker will NOT be started in this process."
)


File Edit Selection View Go Run Terminal Help
Lab8
EXPLORER
LAB8
src
controllers
product_controller.py
report_controller.py
user_controller.py
messaging
__pycache__
tasks
__pycache__
__init__.py
report.py
__init__.py
broker.py
order.py
product.py
taskiq_broker.py
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
main.py
src > main.py
async def lifespan(app: Litestar):
if start_taskiq_in_this_process:
else:
logger.info(
"START_TASKIQ_WORKER not set (or !=1). "
"Taskiq worker will NOT be started in this process."
)
try:
yield
finally:
logger.info("Starting shutdown sequence...")
shutdown_event.set()
for task, name in [(faststream_task, "FastStream"), (taskiq_task, "Taskiq")]:
if task is None:
continue
task.cancel()
try:
await task
except asyncio.CancelledError:
logger.info(f"{name} task cancelled")
except Exception:
logger.exception(f"Error in {name} task")
try:
await redis_client.close()
logger.info("Redis connection closed")
except Exception:
logger.exception("Error closing Redis")
try:
await engine.dispose()
logger.info("Database engine disposed")
except Exception:
logger.exception("Error disposing engine")

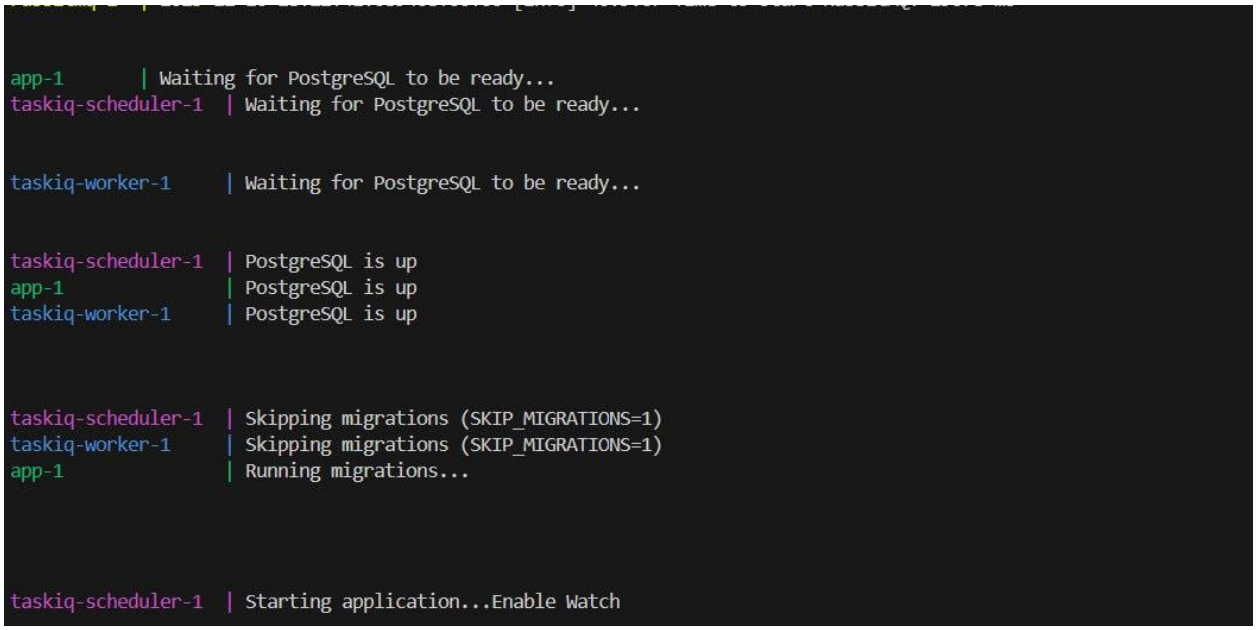
4. Импорт задач в lifespan:

- В блок lifespan, который выполняется при старте приложения, добавлен импорт модуля report из src.messaging.tasks. Импорт модуля report в lifespan гарантирует регистрацию задач при старте приложения
- Сам факт импорта файла report.py приводит к выполнению кода с декоратором @taskiq_broker.task(...). Это "регистрирует" задачу generate_daily_report в системе TaskIQ, после чего планировщик начинает отслеживать её расписание (cron: "0 0 * * *").

После этих изменений приложение становится целостным:

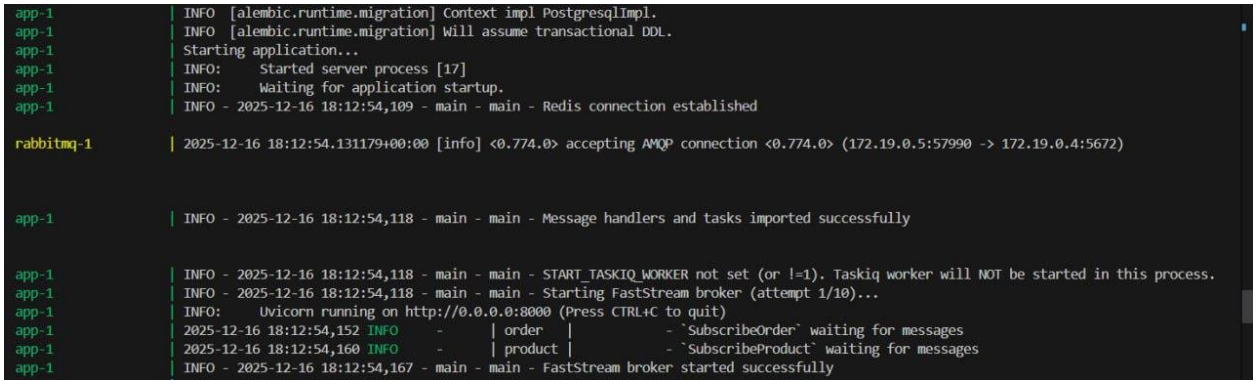
- Пользователь может вручную запрашивать отчёты через REST API (GET /report).
- Система автоматически формирует сводку каждый день в полночь через фоновую задачу TaskIQ.
- Оба сценария используют общую бизнес-логику (ReportService), что исключает дублирование кода и ошибки.

Демонстрация работы системы

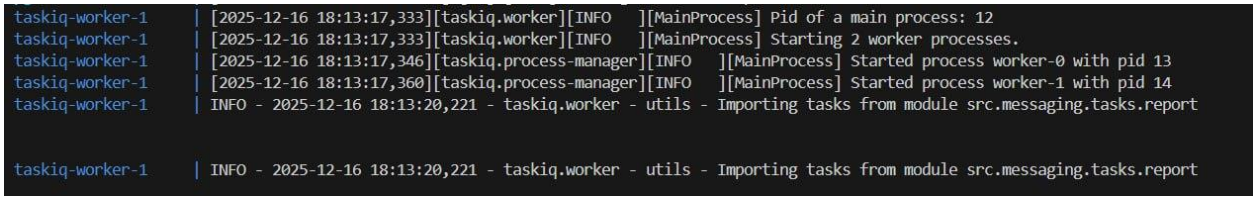


- Все три сервиса (app-1, taskiq-scheduler-1, taskiq-worker-1) ожидают готовности базы данных (PostgreSQL).

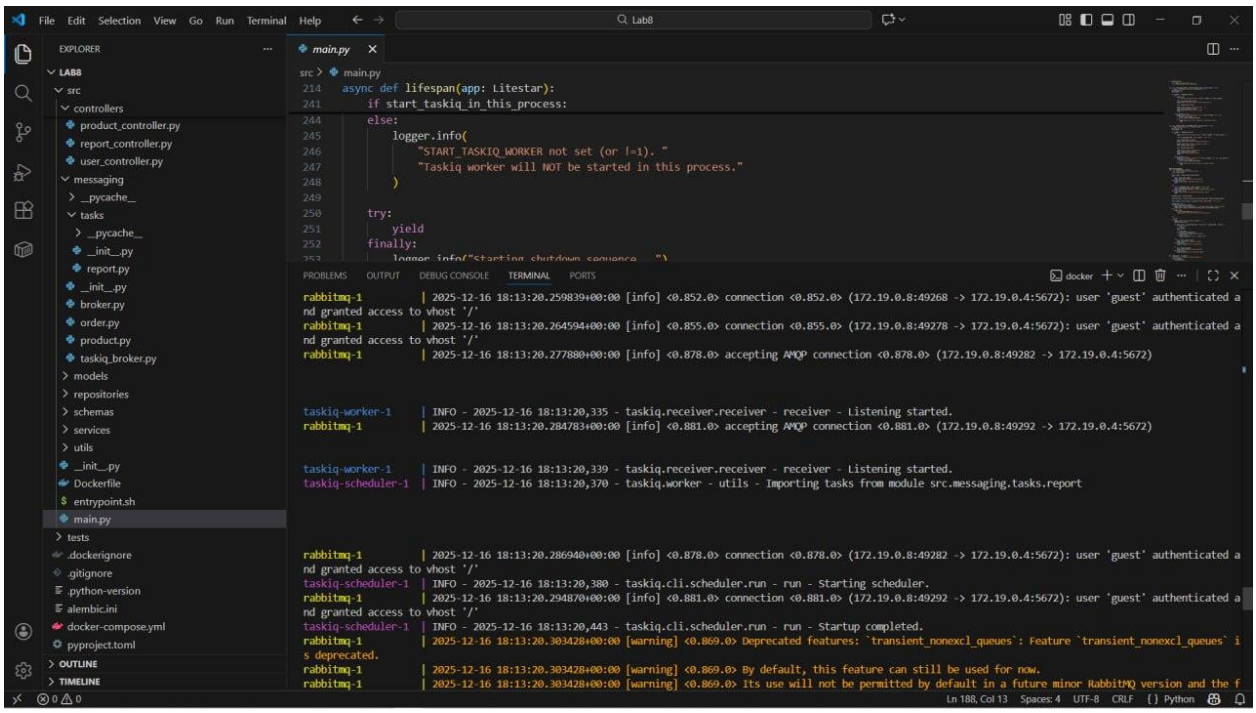
- После запуска БД: основное приложение (app-1) выполняет миграции, изменяя структуру БД. Планировщик и воркер пропускают миграции (Skipping migrations), поскольку используют переменную окружения SKIP_MIGRATIONS="1". Это предотвращает конфликт одновременного доступа к схеме БД.
- Планировщик (taskiq-scheduler-1) завершает инициализацию, сигнализируя, что инфраструктура готова к работе



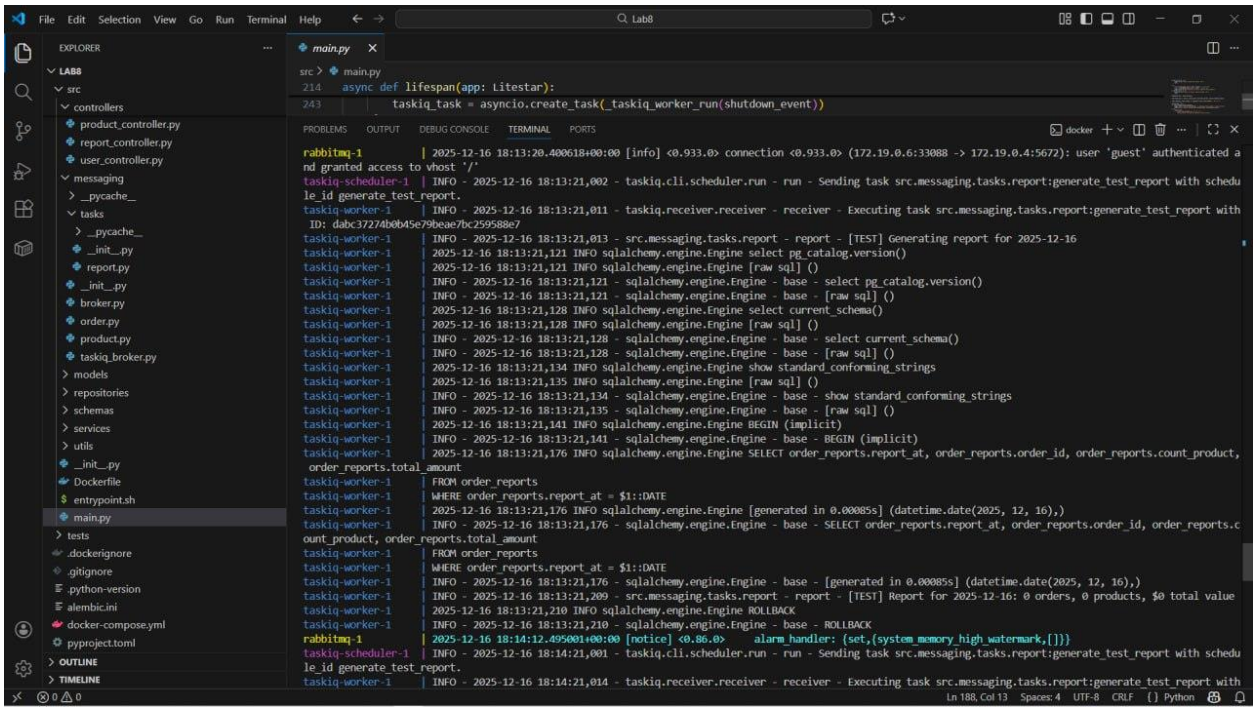
- Главное приложение запустилось (app-1). Подписчики (order, product) ожидают сообщений в RabbitMQ, демонстрируя работу FastStream для обработки событий.



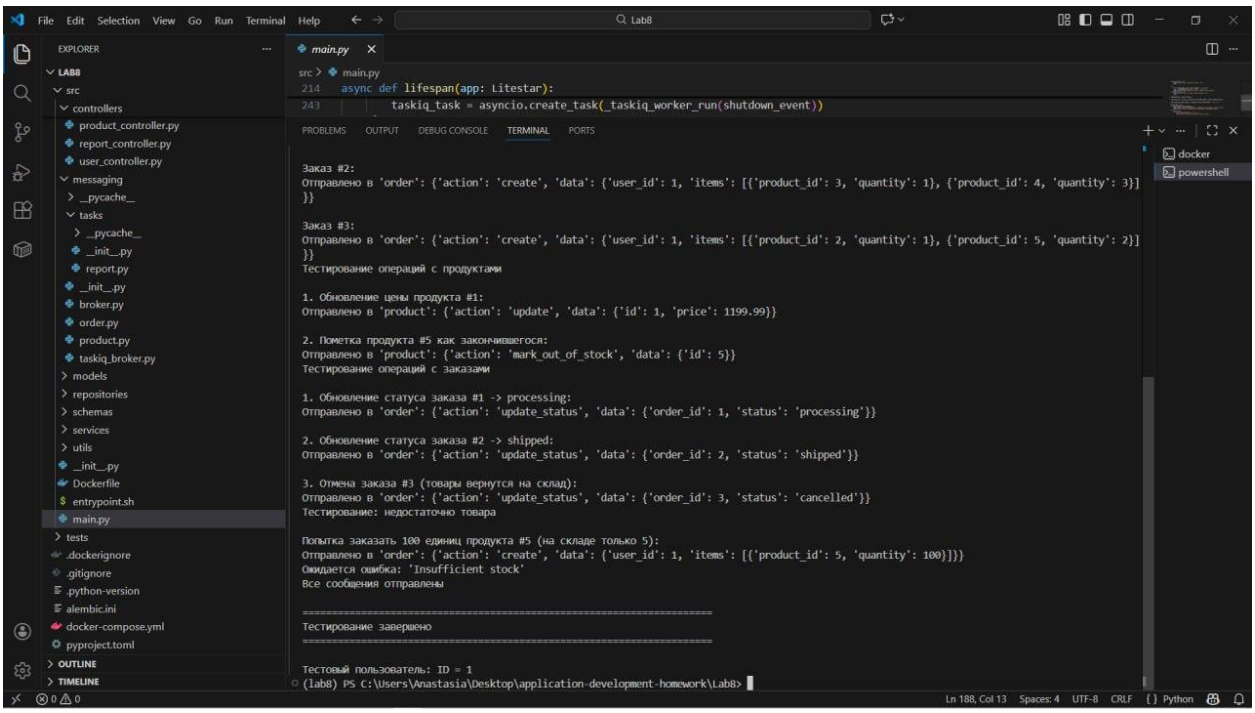
- Запуск воркера TaskIQ (taskiq-worker-1) - запустил 2 рабочих процесса (pid 13 и 14) для параллельного выполнения задач.



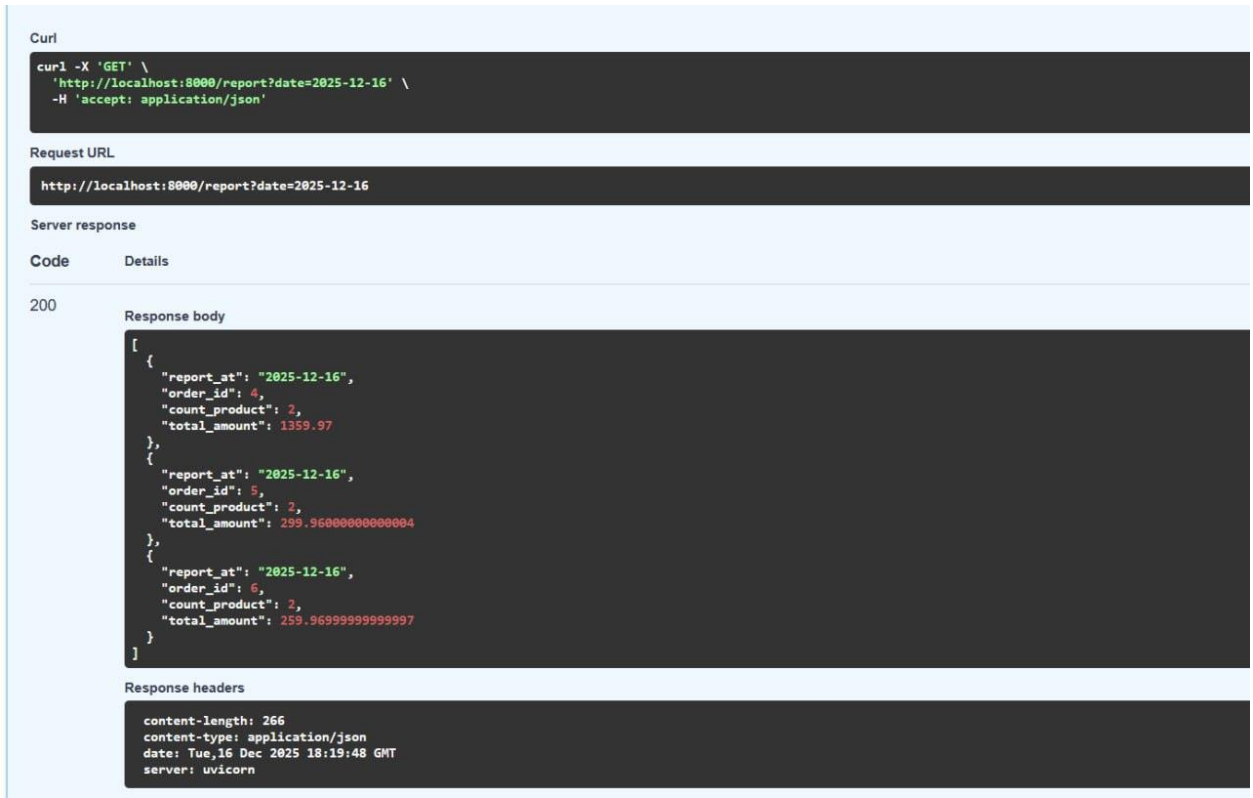
- Воркер (taskiq-worker-1) установил соединения с RabbitMQ и начал прослушивание очереди задач.
- Планировщик (taskiq-scheduler-1) импортировал задачи и успешно запустился, начав отслеживать их расписание.



- Тестовая задача, которая запускается раз в минуту и генерирует отчет запустилась и успешно выполнилась



- Запустили добавление данных через очереди из 6 лабы



- Сформированный отчет за текущий день, задача, которая стоит раз в сутки собирает все заказы из виртуальной таблицы за предыдущие сутки

- Сводный отчет

Ответы на вопросы

1. Что делает Брокер (Broker) в системе Taskiq?

Брокер (Broker) в системе TaskIQ выполняет роль посредника между компонентами системы и отвечает за управление очередью задач. Основные функции брокера:

1) Прием задач - принимает задачи от планировщика или приложения и помещает их в очередь сообщений (в нашем случае RabbitMQ)
2) Распределение задач - передает задачи воркерам для выполнения, обеспечивая балансировку нагрузки между несколькими воркерами
3) Гарантия доставки - обеспечивает надежную доставку задач даже при сбоях, используя механизмы подтверждения (acknowledgment) RabbitMQ
4) Управление состоянием - отслеживает статус выполнения задач (в очереди, выполняется, завершена, ошибка)
5) Сериализация/десериализация - преобразует Python-объекты в формат для передачи по сети и обратно

В проекте используется AioPikaBroker - асинхронный брокер на базе библиотеки aio-pika для работы с RabbitMQ через протокол AMQP.

2. Что делает Планировщик (Scheduler)?

Планировщик (Scheduler) в системе TaskIQ отвечает за управление выполнением задач по расписанию.

Основные функции:

1) Парсинг cron-выражений - анализирует cron-расписания, определенные в декораторах задач (например, "0 0 * * *")
2) Определение времени запуска - вычисляет следующее время выполнения задачи на основе cron-выражения и текущего времени
3) Отправка задач в очередь - в назначенное время отправляет задачу брокеру для выполнения воркером
4) Мониторинг расписаний - постоянно работает в фоне, отслеживая все зарегистрированные расписания
5) Предотвращение дублирования - использует schedule_id для предотвращения повторного запуска одной и той же задачи

В проекте планировщик настроен с LabelScheduleSource, который автоматически находит все задачи с параметром schedule в декораторе @taskiq broker.task(). Планировщик работает как отдельный процесс в Docker-контейнере taskiq-scheduler.

3. Представьте, что ваша задача process_data выполняется в среднем за 10 секунд. Вы настраиваете ее выполнение по cron каждые 5 секунд. К каким проблемам это приведет в долгосрочной перспективе и как это исправить?

Проблемы:

- Накопление задач в очереди - каждые 5 секунд планировщик добавляет

новую задачу, но воркер тратит 10 секунд на выполнение. Это приводит к постоянному росту очереди: через минуту в очереди будет ~6 задач, через час - сотни задач.

- Переполнение памяти - большое количество задач в очереди приводит к высокому потреблению памяти брокером (RabbitMQ) и может вызвать его падение.
- Увеличение задержки - время ожидания выполнения задачи будет постоянно расти, что делает систему неотзывчивой.
- Конкурентное обращение к ресурсам - если несколько экземпляров задачи работают с одними и теми же данными, возможны race conditions и повреждение данных.
- Деградация производительности - постоянная нагрузка может привести к замедлению БД, API и других зависимостей.

Решения:

- Увеличить интервал выполнения - изменить cron на */15 * * * * (каждые 15 секунд) или больше, чтобы задачи успевали завершаться.
- Оптимизировать задачу - ускорить выполнение до <5 секунд через: индексы в БД, кэширование, пакетную обработку, асинхронные операции
- Использовать блокировки - добавить распределенную блокировку (например, через Redis) чтобы предотвратить параллельное выполнение.
- Масштабировать воркеры - запустить несколько воркеров для параллельной обработки, но с осторожностью относительно race conditions.
- Мониторинг очереди — настроить алерты на длину очереди в RabbitMQ и автоматически корректировать частоту запуска.

4. Ваша задача должна запускаться каждый будний день в 9:00 утра по локальному времени офиса (MSK). Какое cron-выражение и параметр

cron_offset вы укажете? В чем подвох, если сервер работает по времени UTC?

Cron-выражение:

```
python@taskiq_broker.task(
   schedule=[
      {
         "cron": "0 9 * * 1-5", # Минута Час День Месяц День_недели
         "cron_offset": 10800, # +3 часа в секундах (MSK = UTC+3)
         "schedule_id": "morning_task_msk",
      }
   ]
)
```

- 0 9 * * 1-5 - 9:00 утра по понедельникам-пятницам
- 1-5 это дни недели, где 1=Пн, 5=Пт
- Параметр cron_offset: 10800 - сдвиг временной зоны (+3 часа = 10800 секунд для MSK)

Подвохи при работе сервера по времени UTC:

1) Переход на летнее/зимнее время в отменен с 2014 года, но, если офис в другой стране, cron_offset не изменится автоматически. Нужно использовать именованные таймзоны, если TaskIQ их поддерживает.
2) Неправильное понимание offset - если сервер в UTC и нужно 9:00 MSK (UTC+3), то: без offset: задача выполнится в 09:00 UTC = 12:00 MSK, а с offset +10800: задача выполнится в 06:00 UTC = 09:00 MSK
3) Проблемы при миграции серверов - при переносе на сервер в другой таймзоне нужно не забыть обновить offset.
4) Логирование - логи будут показывать UTC-время, что может сбивать с толку при отладке.

Лучшая практика:

- Хранить в БД и настройках UTC

- Конвертировать в локальное время только на уровне отображения
- Использовать библиотеки типа pytz или zoneinfo для надежной работы с таймзонами:

5. Масштабирование и отказоустойчивость: если количество задач резко выросло, и один воркер не справляется, какие действия вы предпримете для масштабирования системы на базе Taskiq? Что произойдет, если откажет один из запущенных воркеров? А если откажет планировщик?

Масштабирование при резком росте задач:

- Горизонтальное масштабирование воркеров - добавить больше экземпляров воркеров (реплик). Основной и самый простой путь.
- Автомасштабирование (HPA) - масштабировать количество воркеров по метрикам: длина очереди, latency, CPU/RAM.
- Параметры конкуренции внутри воркера - регулировать число одновременно выполняемых задач (concurrency / prefetch) для оптимального использования ресурсов.
- Вертикальное масштабирование - увеличить CPU/память контейнера, если узким местом являются CPU/IO.
- Оптимизация задач - разбивать тяжёлые задачи на более мелкие, использовать батчинг, кеширование и оптимизацию запросов к БД.
- Приоритизация - выделять отдельные очереди/приоритеты для критичных задач, чтобы важные задания не блокировались.
- Обеспечить, чтобы брокер выдерживал нагрузку (кластеризация, quorum/mirrored queues, персистентность сообщений).

Что произойдёт при отказе одного воркера:

1) Сообщения не теряются: если воркер упал до подтверждения (ack), брокер вернёт сообщение в очередь — другой воркер его подхватит.

2) Задержка выполнения: задача выполнится позже, когда другой воркер

возьмёт сообщение.

3) Возможны частичные побочные эффекты (например, отправлено письмо, но запись в БД не сделана) если задача выполнилась частично до падения.

Меры защиты:

- Идемпотентность задач (проверка, не выполнена ли уже операция).
- Трэйсинг / логирование с уникальным task-id для отладки.
- Механизм повторов с backoff и dead-letter queue для задач, постоянно падающих.
- Таймауты и лимиты на выполнение задач, чтобы упавший процесс не «висел» бесконечно.

При отказе планировщика (scheduler):

1) Задачи по расписанию не будут запускаться - планировщик отвечает за отправку задач в очередь по cron-расписанию. Без него новые задачи не появятся.
2) Уже запущенные задачи продолжат работу - воркеры независимы от планировщика и продолжат обрабатывать очередь.
3) Пропущенные задачи - если планировщик был недоступен в момент запланированного времени, задача будет пропущена (по умолчанию).
4) Решения: резервный планировщик (запустить 2 экземпляра scheduler с использованием distributed lock (например, через Redis), чтобы только один был активен); Catch-up механизм (настроить TaskIQ на выполнение пропущенных задач).

Вывод:

В ходе выполнения лабораторной работы была успешно реализована система автоматической генерации отчётов на основе TaskIQ: создано SQL-представление order_reports, реализована многослойная архитектура

(Model/Repository/Service/Controller), настроен TaskIQ-scheduler с ежедневным cron, добавлены REST-API и контейнеризация в Docker. Достигнуты цели: автоматическая генерация отчётов, отказоустойчивость и готовность к масштабированию. В процессе освоены асинхронные задачи, настройка cron, создание SQL-представлений и проектирование распределённых систем — решение пригодно для использования в production.