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 №6

Основы работы с RabbitMQ

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

Екатеринбург 2025

Цель работы

Освоить основные принципы работы с брокером сообщений RabbitMQ в Python, изучить различные паттерны обмена сообщениями и научиться создавать распределенные приложения.

Задачи

1) Настройка инфраструктуры RabbitMQ

- Добавить сервис RabbitMQ в Docker Compose с образом rabbitmq:3-management
- Настроить порты для AMQP протокола (5672) и веб-интерфейса управления (15672)
- Обеспечить персистентность данных через volume и настроить healthcheck
- Интегрировать RabbitMQ с существующим приложением через переменные окружения

2) Реализация обработчиков очередей

- Создать подписчики для очередей order и product с использованием FastStream
- Реализовать обработку операций создания, обновления продукции и пометки товаров как недоступных
- Реализовать обработку создания заказов с валидацией остатков, обновления статусов и возврата товаров при отмене
- Обеспечить транзакционную обработку сообщений с логированием и обработкой ошибок

3) Интеграция брокера с жизненным циклом приложения

- Встроить запуск и остановку RabbitBroker в lifespan контекст Litestar
- Реализовать retry логику с автоматическими повторными попытками подключения к брокеру
- Обеспечить graceful shutdown с корректным закрытием соединений при остановке приложения

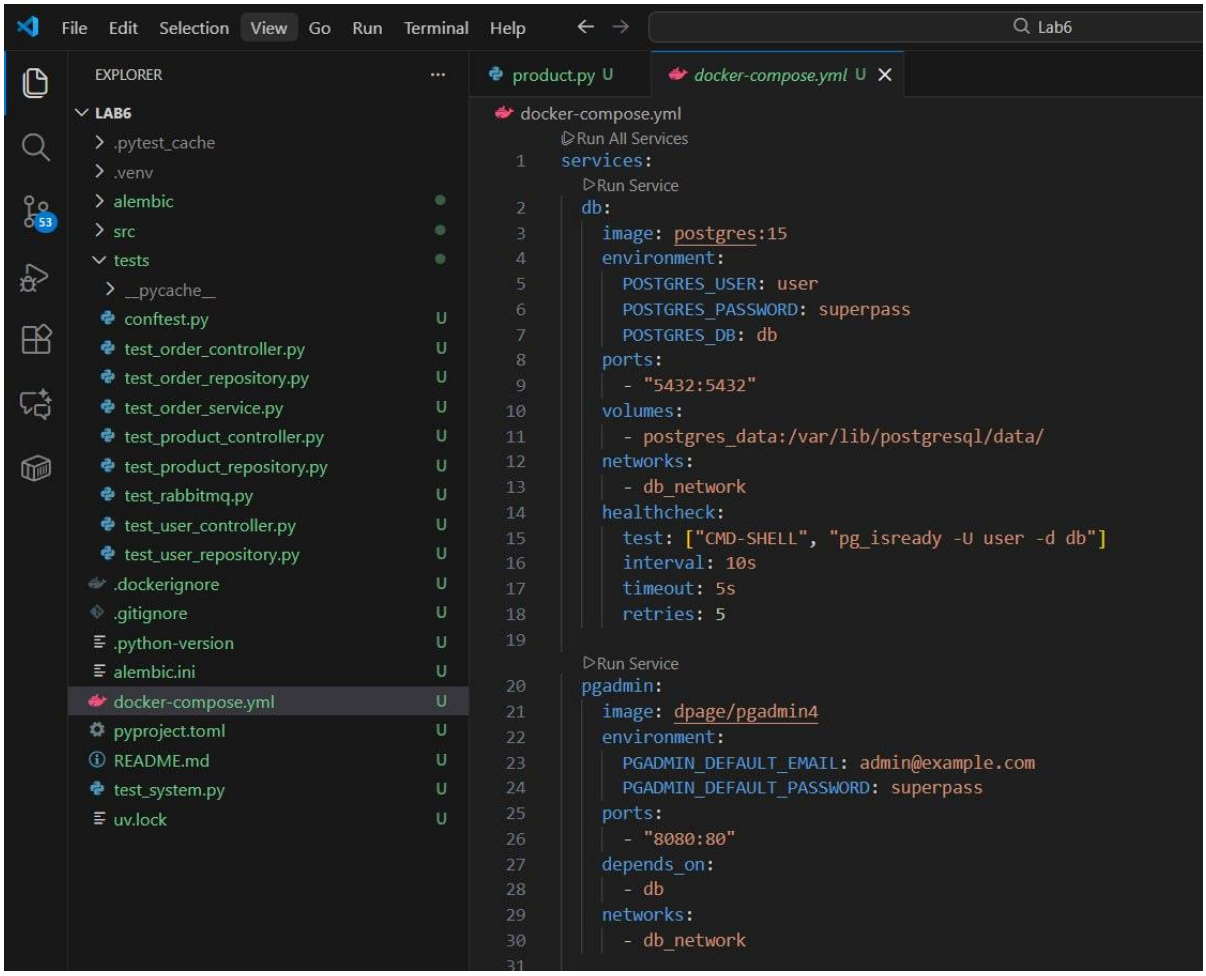
- Разделить ответственность: команды через RabbitMQ, запросы через REST API

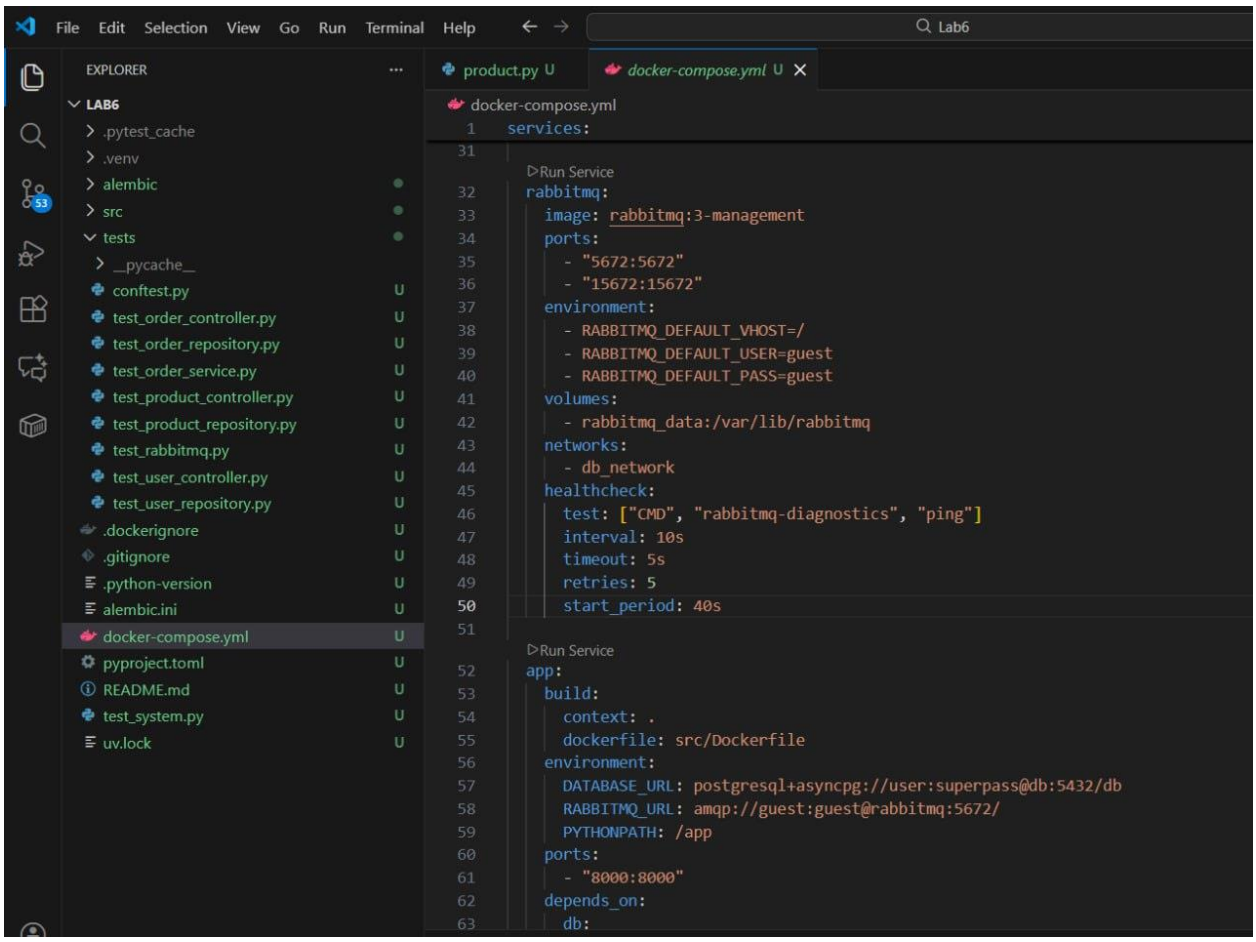
4) Разработка продюсера и тестирование системы

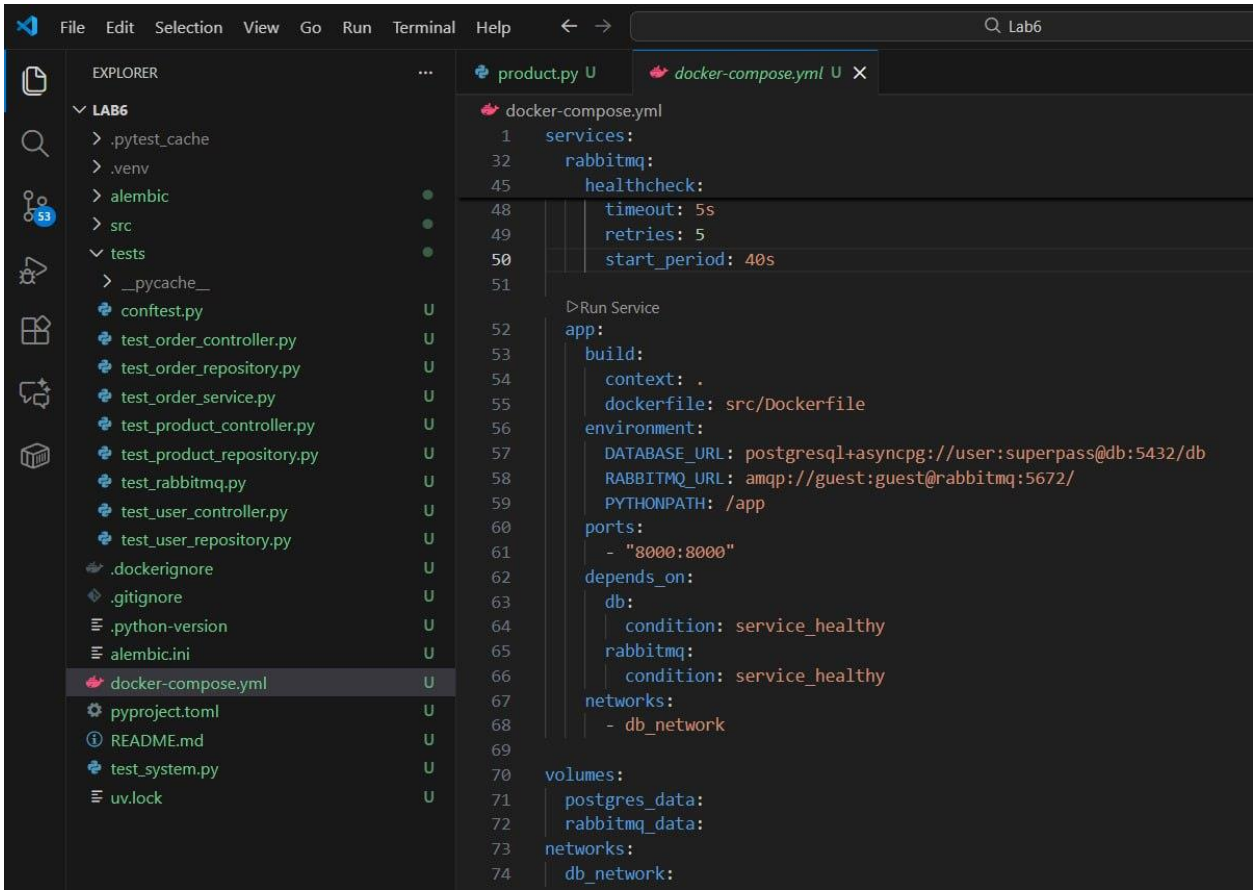
- Создать скрипт-продюсер с использованием библиотеки aio-pika для отправки сообщений
- Реализовать создание 5 тестовых продуктов и 3 заказов с различными параметрами
- Протестировать все операции системы: создание, обновление, изменение статусов, пометку товаров
- Проверить обработку граничных случаев (недостаточное количество товара, отмена заказа)

Часть 1: Настройка контейнера с RabbitMQ

Файл: *docker-compose.yml*







Ход выполнения:

Файл содержит конфигурацию сервисов

- PostgreSQL
- RabbitMQ с Management UI
- app

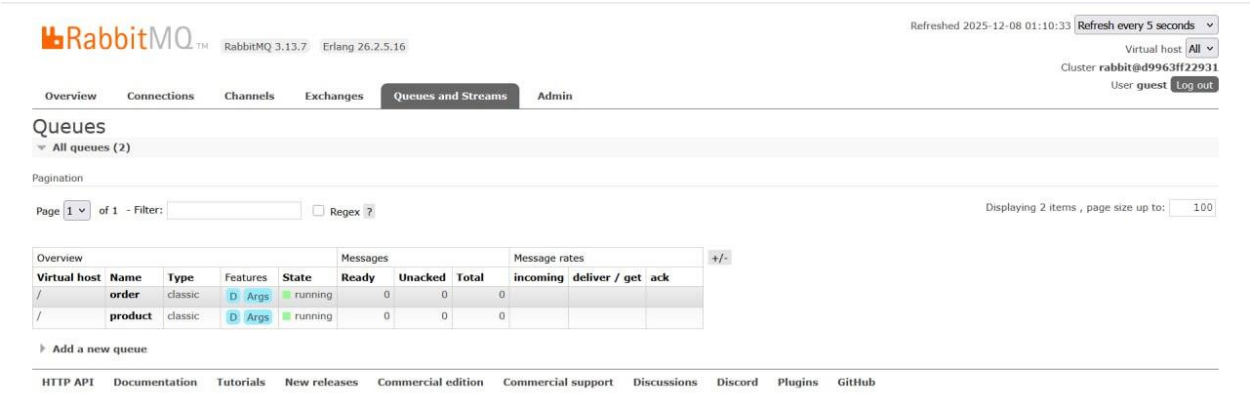
Был добавлен сервис rabbitmq с образом rabbitmq:3-management:

- Порты: 5672 (AMQP протокол), 15672 (веб-интерфейс управления)
- Переменные окружения для дефолтного vhost и учетных данных (guest/guest)
- Volume rabbitmq_data для персистентности данных
- Healthcheck с использованием rabbitmq-diagnostics ping для проверки готовности сервиса

Сервис app обновлен:

- Добавлена переменная RABBITMQ_URL для подключения к брокеру
- Зависимость от rabbitmq с условием service_healthy — приложение стартует только после готовности RabbitMQ

По адресу http://localhost:15672 проверяем работу RabbitMQ



Файл: *pyproject.toml*

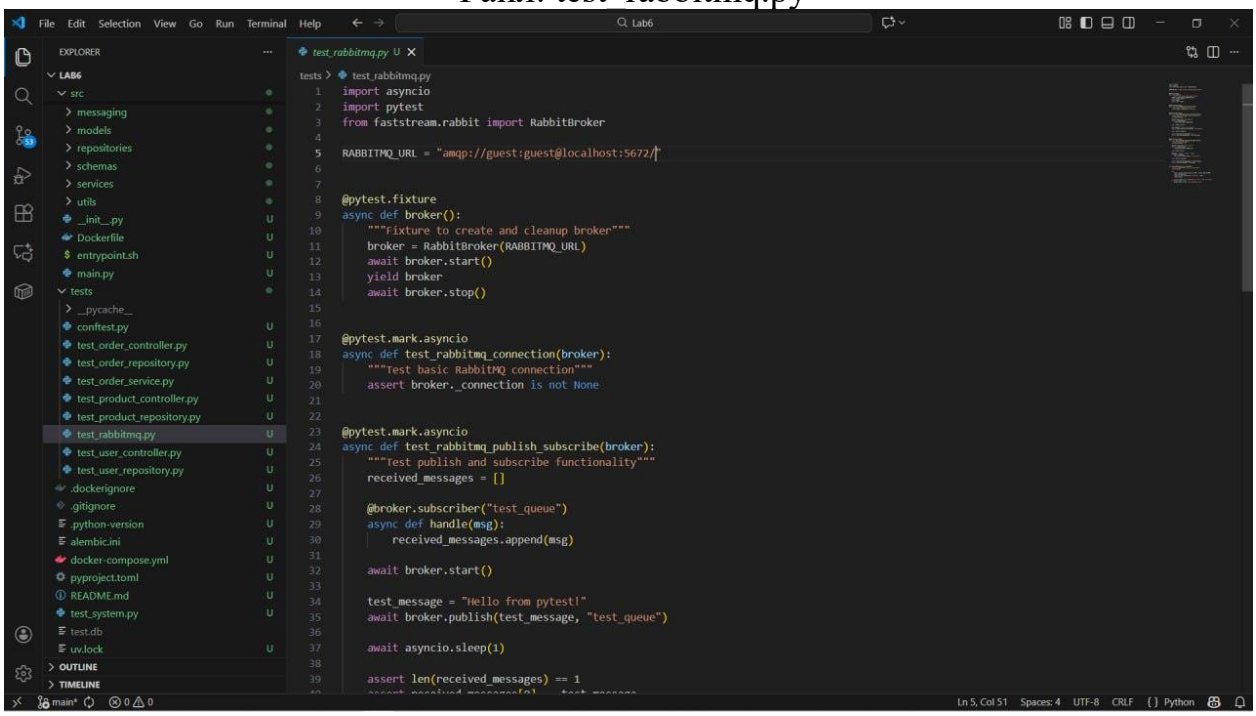
```toml
[project]
name = "lab6"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
    "aio-pika>=9.4.3",
    "aiosqlite>=0.21.0",
    "alembic>=1.17.2",
    "asyncpg>=0.30.0",
    "black>=25.11.0",
    "fast-depends>=3.0.5",
    "faststream[rabbit]>=0.1.4",
    "isort>=7.0.0",
    "litestar[standard]>=2.18.0",
    "polyfactory>=3.0.0",
    "pre-commit>=4.5.0",
    "pydantic[email]>=2.12.4",
    "pylint>=4.0.3",
    "pytest>=9.0.1",
    "pytest-asyncio>=1.3.0",
    "sniffio>=1.3.1",
    "sqlalchemy[asyncio]>=2.0.44",
    "uvicorn[standard]>=0.38.0",
]
```

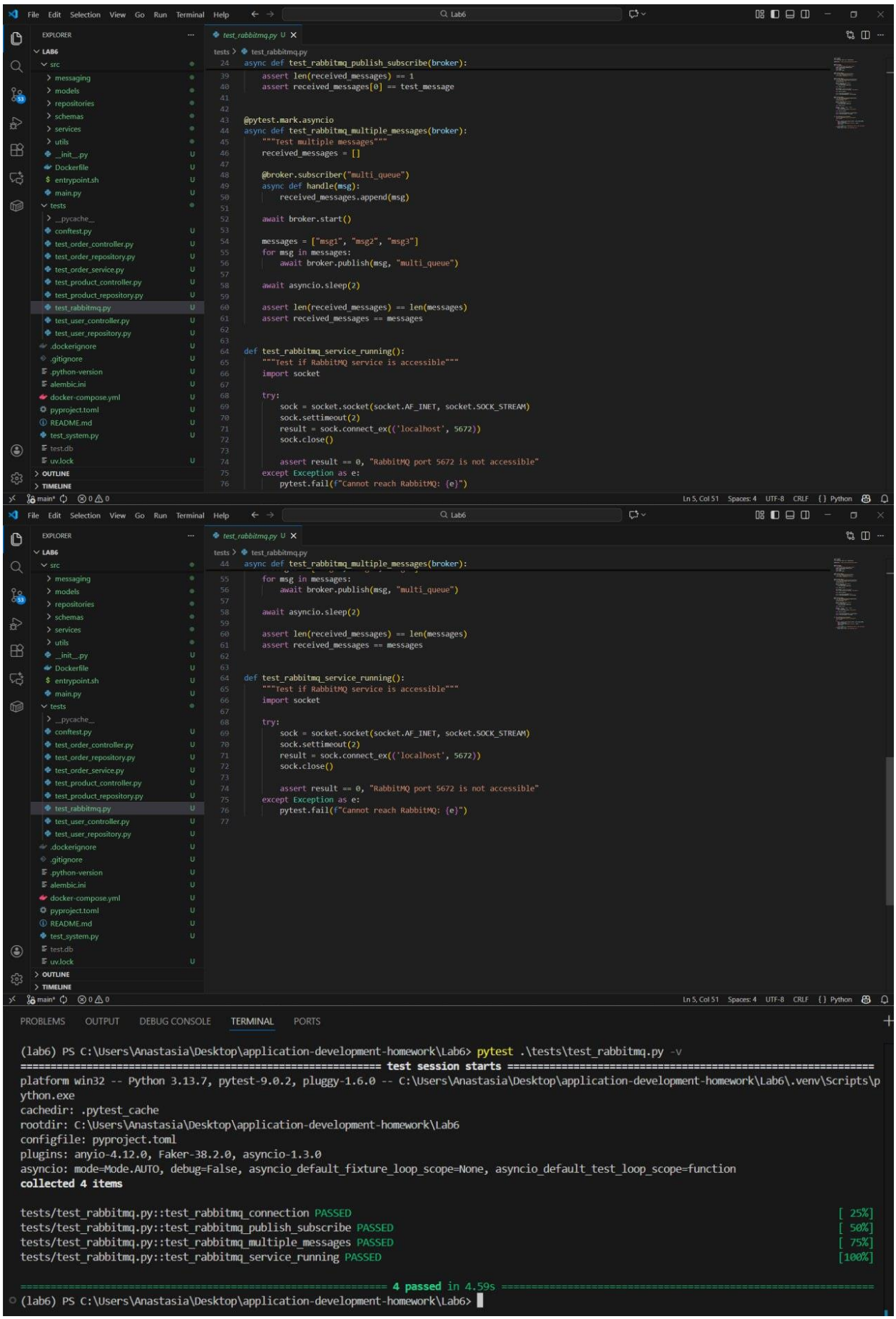
Установлены библиотеки для работы с RabbitMQ:

- faststream[rabbit] — фреймворк для асинхронной обработки сообщений
- aio-pika — асинхронный клиент AMQP для отправки сообщений из продюсера

Часть 2: Подключение к очередям

Файл: test_rabbitmq.py





Файл содержит юнит-тесты для проверки базовой работы RabbitMQ:

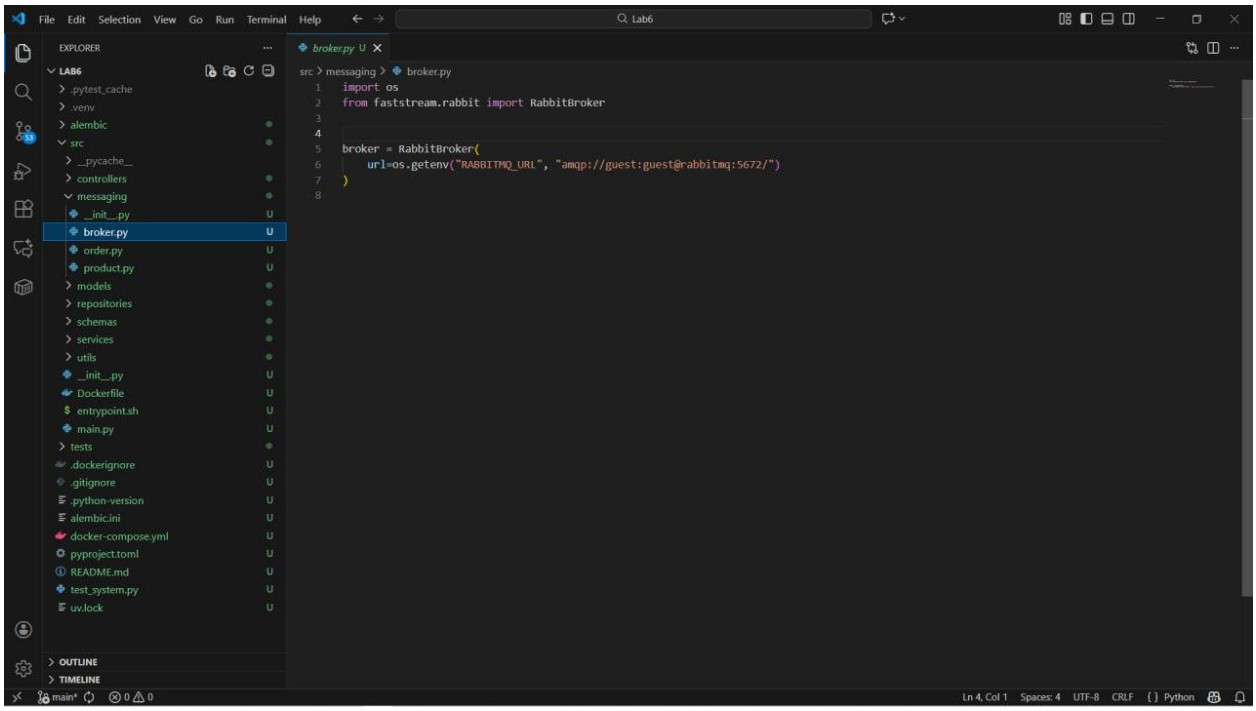
- test_rabbitmq_connection — проверка установки соединения
- test_rabbitmq_publish_subscribe — отправка и получение одного сообщения
- test_rabbitmq_multiple_messages — обработка нескольких сообщений подряд
- test_rabbitmq_service_running — проверка доступности порта 5672

Используется фикстура broker для создания и очистки тестового брокера.

Часть 3: Реализация обработчиков очередей

Файл: *broker.py*
(Подключение к очередям в приложении)

Ход выполнения:



Файл broker.py содержит конфигурацию подключения - создание экземпляра брокера с URL.

В нем создается глобальный экземпляр RabbitBroker, который:

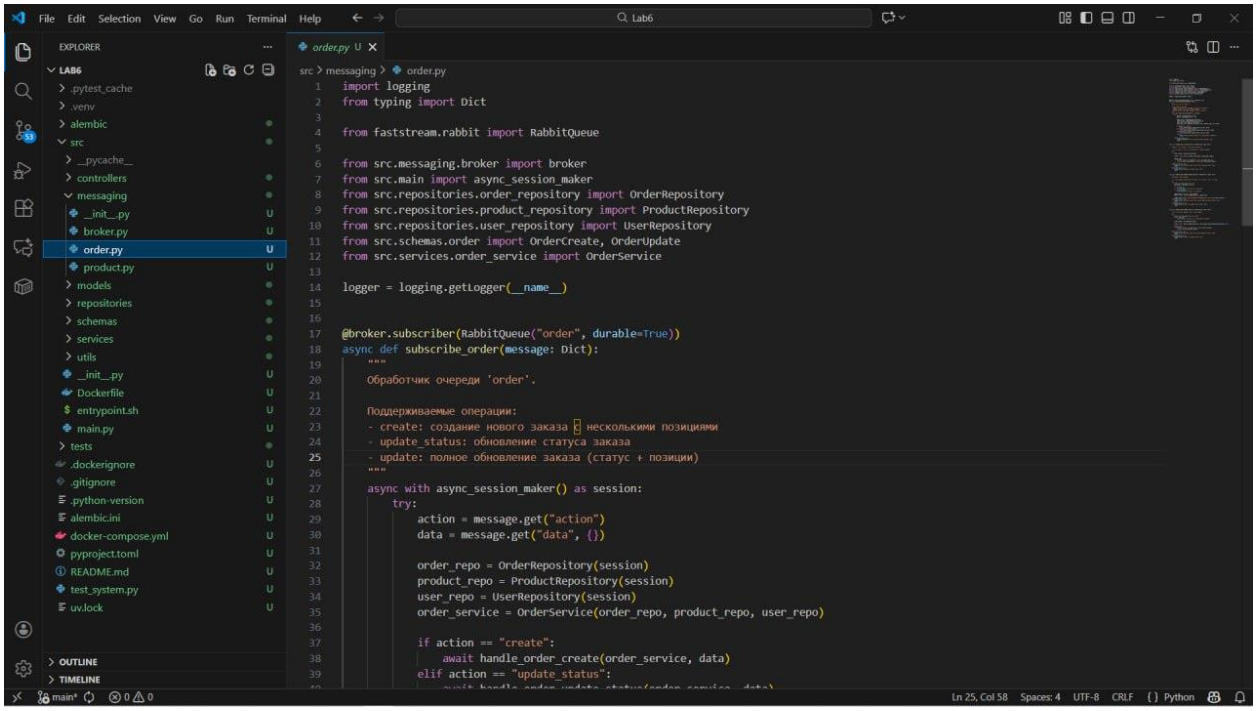
- Читает URL подключения из переменной окружения RABBITMQ_URL
- Используется для подписки на очереди и публикации сообщений

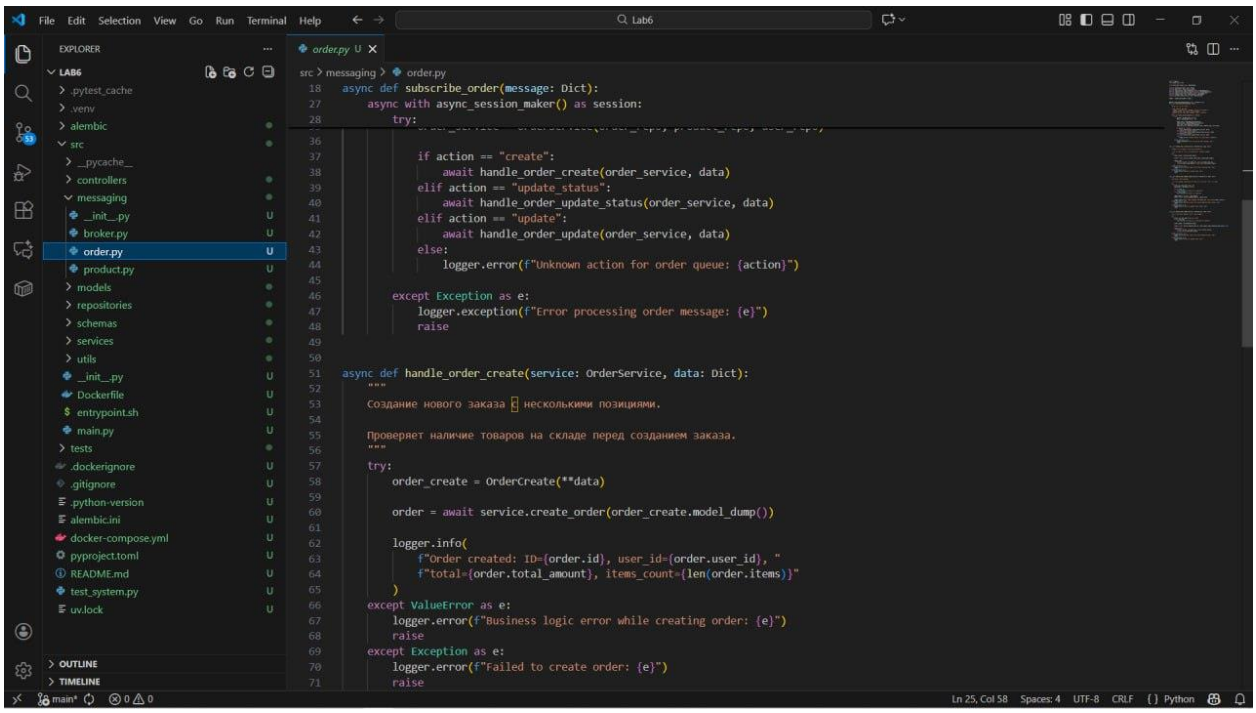
- Управляет соединением с RabbitMQ через устойчивое соединение

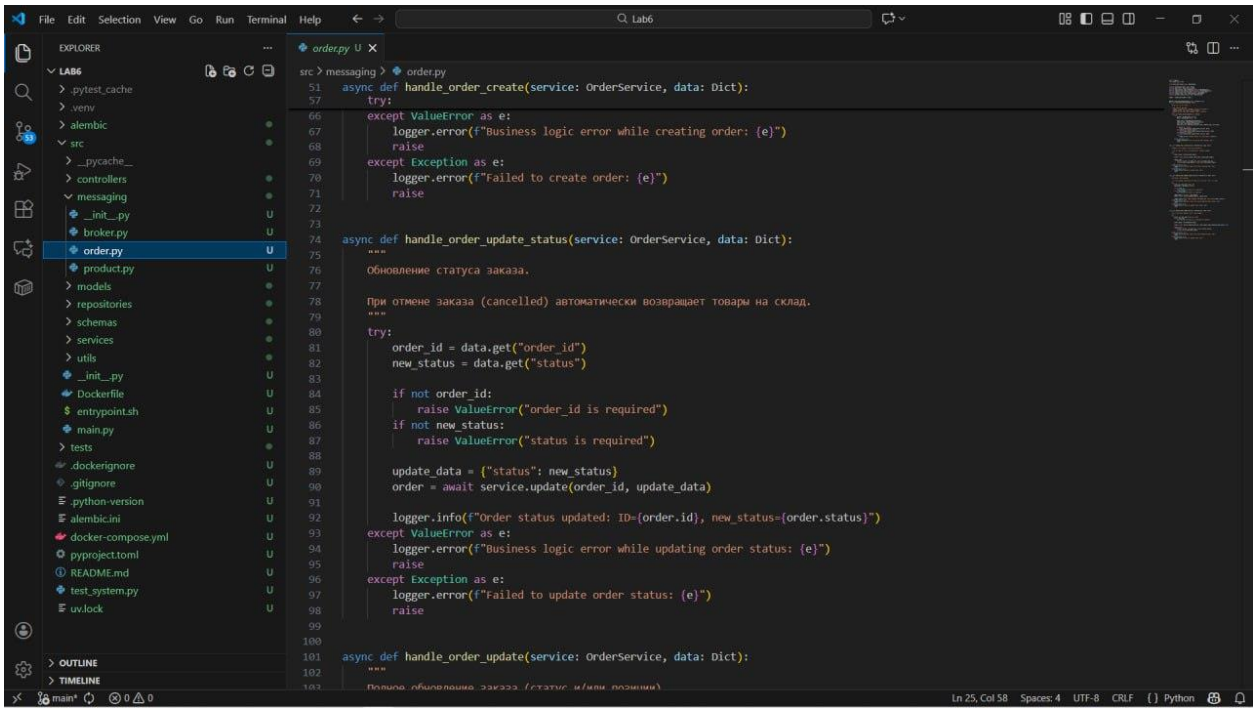
Подключение к очередям в приложении также происходит в несколько этапов**:**

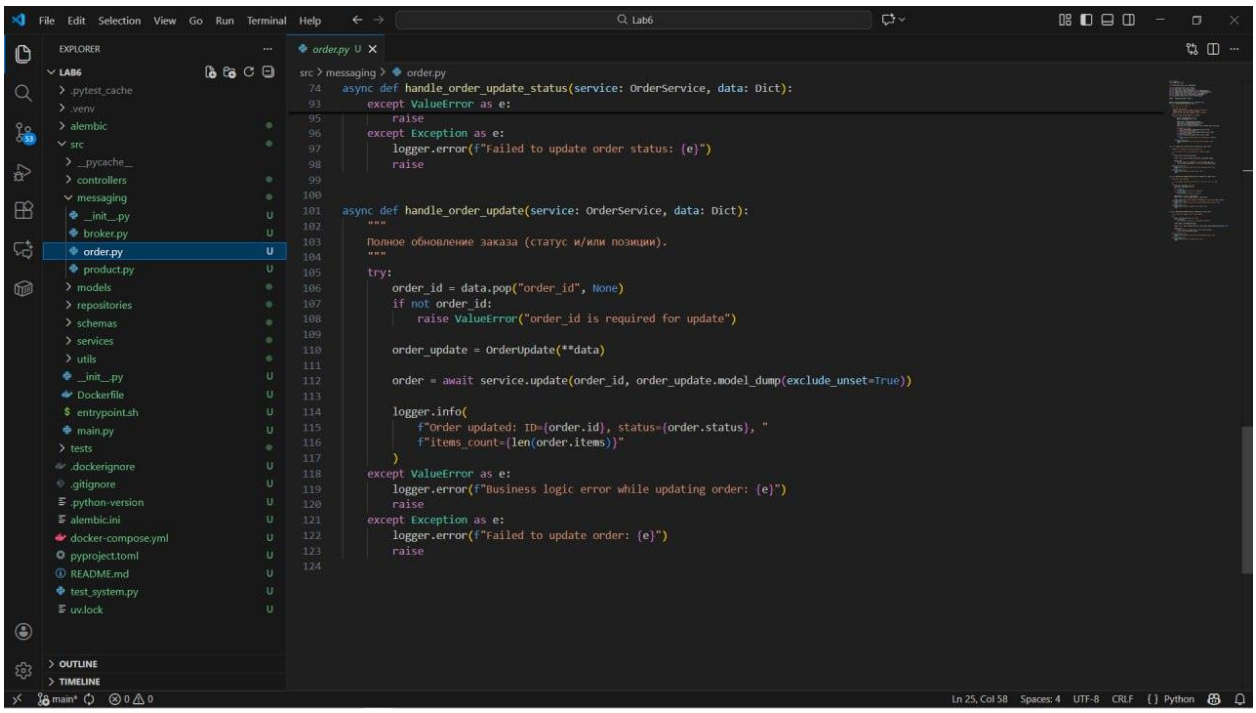
1. **Декларация подписчиков в файлах: src/messaging/order.py и src/messaging/product.py.** В них происходит обработка события из очереди продукта и из очереди заказов.
2. **Активация подключения в файле *main.py***

Файл: *src/messaging/order.py*




File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
order.py U
src > messaging > order.py
async def subscribe_order(message: Dict):
async with async_session_maker() as session:
try:
if action == "create":
await handle_order_create(order_service, data)
elif action == "update_status":
await handle_order_update_status(order_service, data)
elif action == "update":
await handle_order_update(order_service, data)
else:
logger.error(f"Unknown action for order queue: {action}")
except Exception as e:
logger.exception(f"Error processing order message: {e}")
raise
async def handle_order_create(service: OrderService, data: Dict):
"""
Создание нового заказа с несколькими позициями.
Проверяет наличие товаров на складе перед созданием заказа.
"""
try:
order_create = OrderCreate(**data)
order = await service.create_order(order_create.model_dump())
logger.info(
f"Order created: ID={order.id}, user_id={order.user_id}, "
f"total={order.total_amount}, items_count={len(order.items)}"
)
except ValueError as e:
logger.error(f"Business logic error while creating order: {e}")
raise
except Exception as e:
logger.error(f"Failed to create order: {e}")
raise
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
__init__.py
broker.py
order.py
product.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
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 25, Col 58
Spaces: 4
UTF-8
CRLF
{} Python


File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
order.py U
src > messaging > order.py
async def handle_order_create(service: OrderService, data: Dict):
try:
except ValueError as e:
logger.error(f"Business logic error while creating order: {e}")
raise
except Exception as e:
logger.error(f"Failed to create order: {e}")
raise
async def handle_order_update_status(service: OrderService, data: Dict):
"""
Обновление статуса заказа.
При отмене заказа (cancelled) автоматически возвращает товары на склад.
"""
try:
order_id = data.get("order_id")
new_status = data.get("status")
if not order_id:
raise ValueError("order_id is required")
if not new_status:
raise ValueError("status is required")
update_data = {"status": new_status}
order = await service.update(order_id, update_data)
logger.info(f"Order status updated: ID={order.id}, new_status={order.status}")
except ValueError as e:
logger.error(f"Business logic error while updating order status: {e}")
raise
except Exception as e:
logger.error(f"Failed to update order status: {e}")
raise
async def handle_order_update(service: OrderService, data: Dict):
"""
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
__init__.py
broker.py
order.py
product.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
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 25, Col 58
Spaces: 4
UTF-8
CRLF
{} Python



В файле реализован обработчик очереди order с 3 операциями:

1. create — создание заказа:
   - Валидирует входные данные через OrderCreate схему
   - Проверяет наличие товаров на складе через OrderService.create_order()
   - Автоматически списывает товары при успешном создании
   - Логирует ID заказа, пользователя, сумму и количество позиций

2. update_status — изменение статуса:
   - Принимает order_id и новый status
   - При статусе cancelled автоматически возвращает товары на склад (логика в сервисе)
   - Поддерживает статусы: pending, processing, shipped, delivered, cancelled

3. update — полное обновление:
   - Позволяет изменить статус и/или позиции заказа
   - Использует OrderUpdate схему с опциональными полями

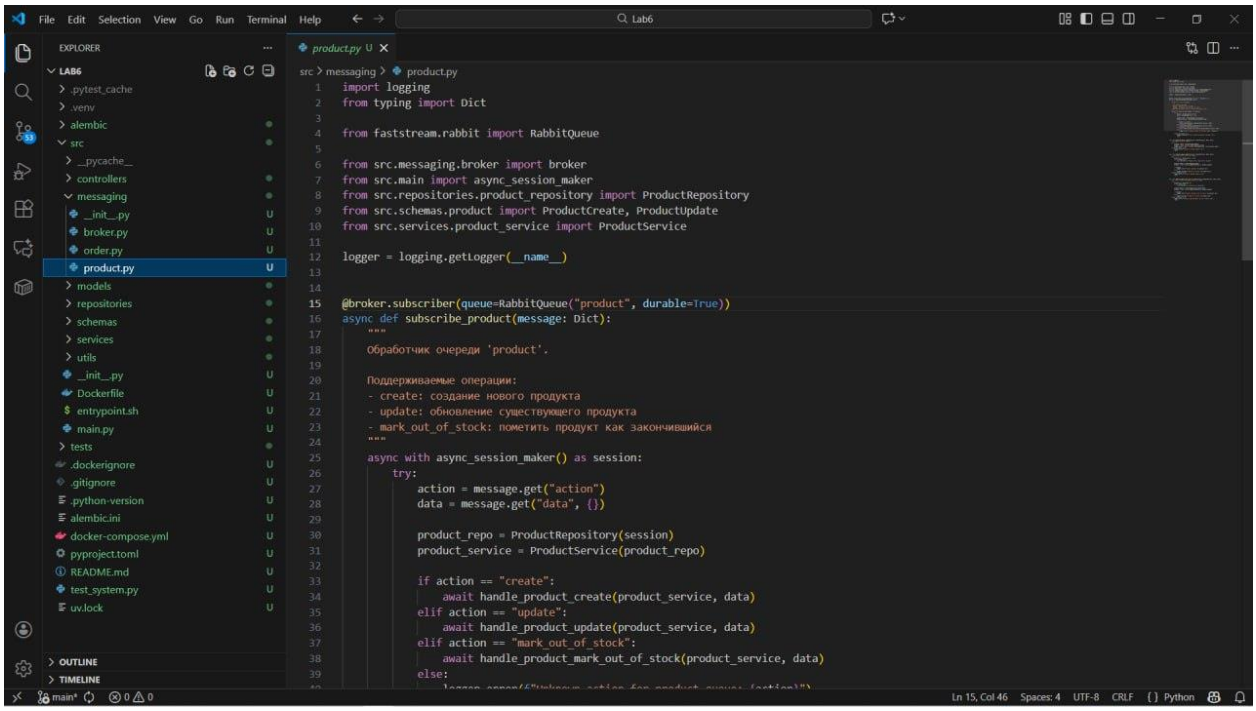
Также добавлена обработка ошибок:

- ValueError — бизнес-логика (недостаточно товара, неверные

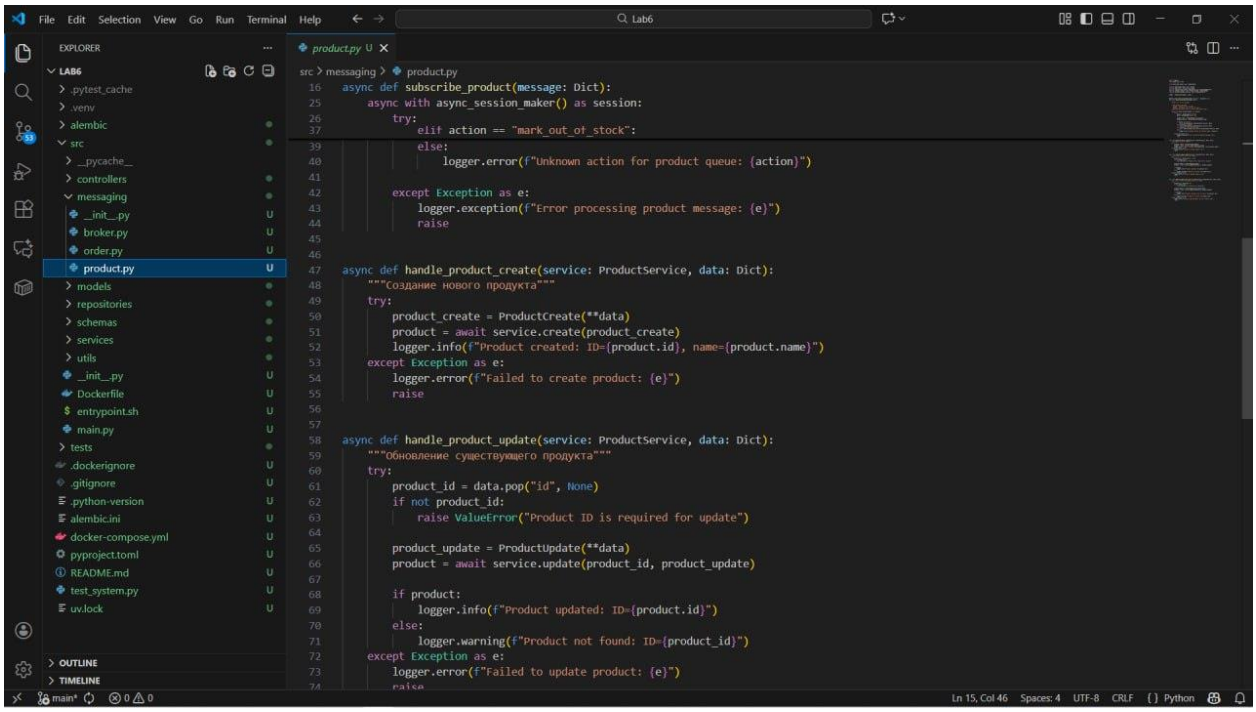
параметры)

- Все исключения логируются и пробрасываются для повторной обработки RabbitMQ

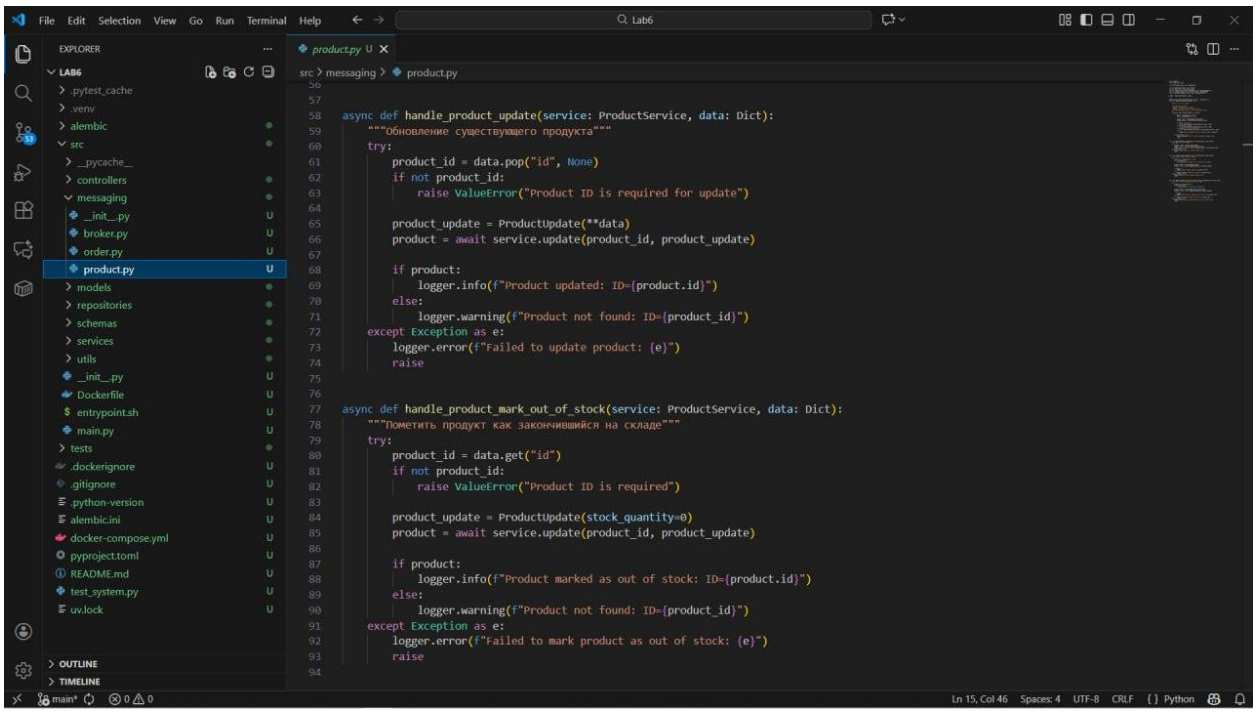
Файл: *src/messaging/product.py*



```python
import logging
from typing import Dict

from faststream.rabbit import RabbitQueue

from src.messaging.broker import broker
from src.main import async_session_maker
from src.repositories.product_repository import ProductRepository
from src.schemas.product import ProductCreate, ProductUpdate
from src.services.product_service import ProductService

logger = logging.getLogger(__name__)


@broker.subscriber(queue=RabbitQueue("product", durable=True))
async def subscribe_product(message: Dict):
    """
    Обработчик очереди 'product'.

    Поддерживаемые операции:
    - create: создание нового продукта
    - update: обновление существующего продукта
    - mark_out_of_stock: пометить продукт как закончившийся
    """
    async with async_session_maker() as session:
        try:
            action = message.get("action")
            data = message.get("data", {})

            product_repo = ProductRepository(session)
            product_service = ProductService(product_repo)

            if action == "create":
                await handle_product_create(product_service, data)
            elif action == "update":
                await handle_product_update(product_service, data)
            elif action == "mark_out_of_stock":
                await handle_product_mark_out_of_stock(product_service, data)
            else:
```



```python
async def subscribe_product(message: Dict):
    async with async_session_maker() as session:
        try:
            elif action == "mark_out_of_stock":
            else:
                logger.error(f"Unknown action for product queue: {action}")

        except Exception as e:
            logger.exception(f"Error processing product message: {e}")
            raise


async def handle_product_create(service: ProductService, data: Dict):
    """Создание нового продукта"""
    try:
        product_create = ProductCreate(**data)
        product = await service.create(product_create)
        logger.info(f"Product created: ID={product.id}, name={product.name}")
    except Exception as e:
        logger.error(f"Failed to create product: {e}")
        raise


async def handle_product_update(service: ProductService, data: Dict):
    """Обновление существующего продукта"""
    try:
        product_id = data.pop("id", None)
        if not product_id:
            raise ValueError("Product ID is required for update")

        product_update = ProductUpdate(**data)
        product = await service.update(product_id, product_update)

        if product:
            logger.info(f"Product updated: ID={product.id}")
        else:
            logger.warning(f"Product not found: ID={product_id}")
    except Exception as e:
        logger.error(f"Failed to update product: {e}")
        raise
```



Реализован обработчик очереди product с 3 операциями:

1. create — создание продукта:
   - Валидация через ProductCreate (name, price, stock_quantity)
   - Сохранение в БД через ProductService.create()

2. update — обновление продукта:
   - Требует id в данных
   - Частичное обновление через ProductUpdate (цена, количество на складе)

3. mark_out_of_stock — пометка как недоступного:
   - Устанавливает stock_quantity = 0
   - Используется для явного обозначения отсутствия товара

Также в файлах: src/messaging/order.py и src/messaging/product.py происходит объявление обработчиков через декораторы:

- @broker.subscriber(RabbitQueue("order", durable=True))
  async def subscribe_order(message: Dict):

- @broker.subscriber(queue=RabbitQueue("product", durable=True))
  async def subscribe_product(message: Dict):

Декоратор @broker.subscriber регистрирует функцию как обработчик очереди, но не запускает подключение.Оба обработчика используют:

- RabbitQueue(..., durable=True) — очередь сохраняется при перезапуске RabbitMQ
- Создание новой сессии БД для каждого сообщения через async_session_maker
- Логирование всех операций с деталями

2. Активация подключения

Файл: *main.py*

(изменения в main для поддержки очередей)

```python
async def _broker_connect_loop(shutdown_event: asyncio.Event) -> None:
    """
    Фоновая задача для запуска брокера и поддержания его работы до завершения работы системы.
    Брокер самостоятельно управляет переподключением благодаря своему устойчивому соединению.
    """
    max_retries = 10
    retry_delay = 5

    for attempt in range(max_retries):
        try:
            logger.info(f"Attempting to start Rabbit broker (attempt {attempt + 1}/{max_retries})...")
            await broker.start()
            logger.info("Rabbit broker connected successfully")

            await shutdown_event.wait()

            logger.info("Shutdown requested, closing broker...")
            await broker.close()
            logger.info("Broker closed successfully")
            return

        except Exception as e:
            logger.error(f"Broker connection error (attempt {attempt + 1}/{max_retries}): {e}")
            if attempt < max_retries - 1:
                logger.info(f"Retrying in {retry_delay} seconds...")
                await asyncio.sleep(retry_delay)
            else:
                logger.error("Max retries reached, giving up on broker connection")
                raise
```

```python
@asynccontextmanager
async def lifespan(app: Litestar):
    """
    Управление жизненным циклом приложения.

    Выполняется при старте и остановке приложения.
    """
    shutdown_event = asyncio.Event()
    broker_task = asyncio.create_task(_broker_connect_loop(shutdown_event))

    from src.messaging import order, product # noqa: F401

    try:
        yield
    finally:
        shutdown_event.set()

        try:
            await broker.close()
        except Exception:
            logger.exception("Error while closing broker")

        broker_task.cancel()
        try:
            await broker_task
        except asyncio.CancelledError:
            pass
        except Exception:
            logger.exception("Broker task raised while shutting down")

        try:
            await engine.dispose()
        except Exception:
            logger.exception("Error disposing engine")

```

Интеграция с приложением

В функцию lifespan добавлен жизненный цикл брокера:

_broker_connect_loop() — фоновая задача:

- Пытается подключиться к RabbitMQ до 10 раз с задержкой 5 секунд
- Вызывает broker.start() для начала прослушивания очередей
- Ожидает сигнала завершения через shutdown_event
- Корректно закрывает соединение при остановке приложения

Импорт обработчиков: pythonfrom src.messaging import order, product. Необходим для регистрации декораторов @broker.subscriber до старта брокера

Здесь происходит подключение к RabbitMQ и начинается прослушивание всех зарегистрированных очередей

- async def _broker_connect_loop (shutdown_event: asyncio.Event):

await broker.start()

Graceful shutdown:

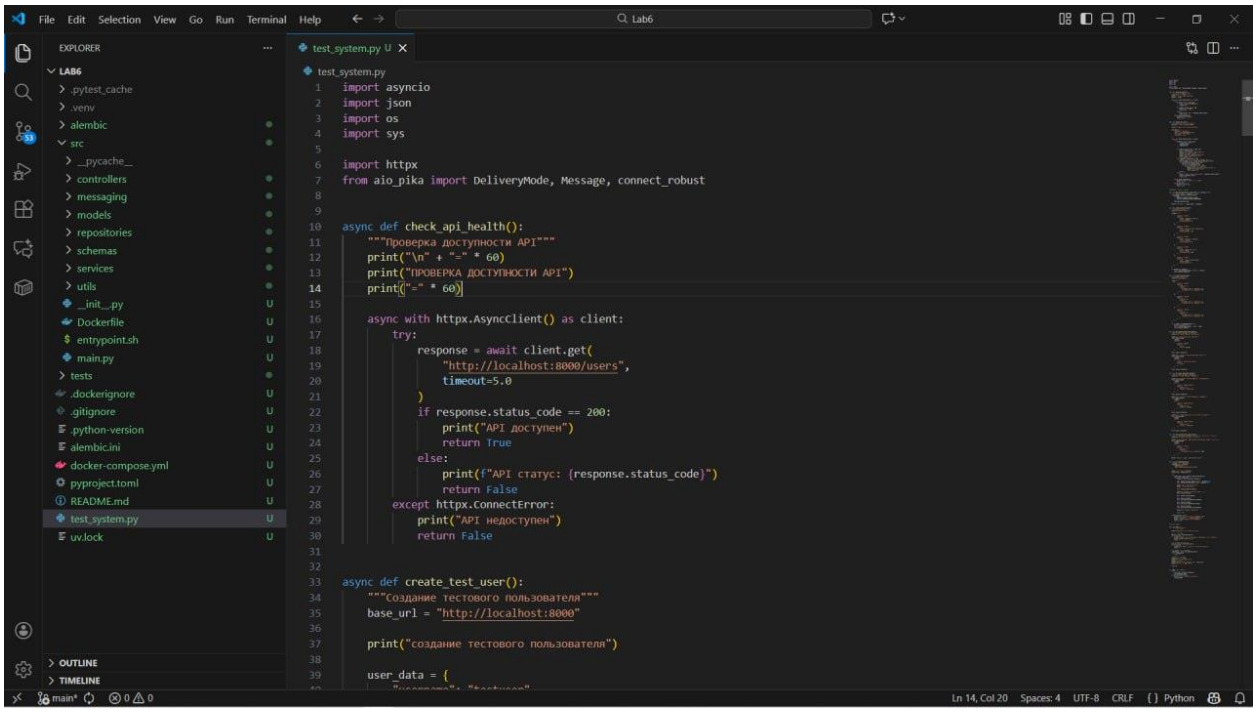
- Устанавливает флаг shutdown_event
- Отменяет фоновую задачу брокера
- Закрывает соединение с БД через engine.dispose()

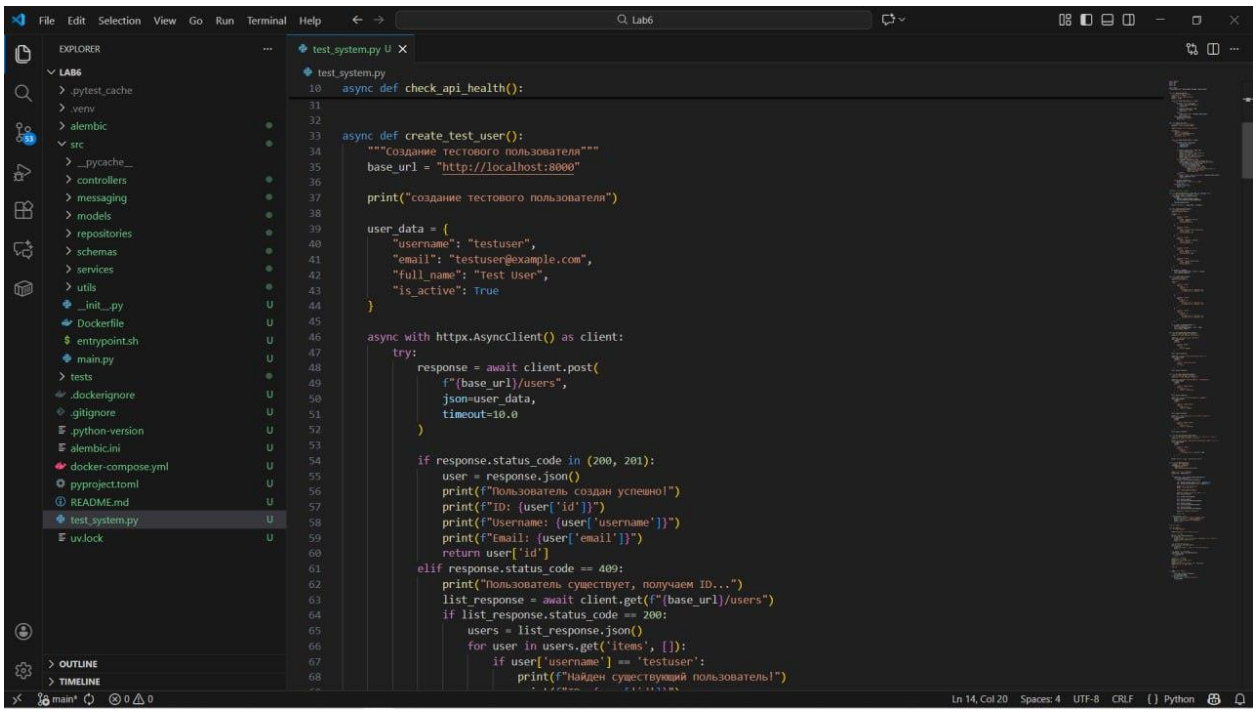
Часть 4: Продюсер данных

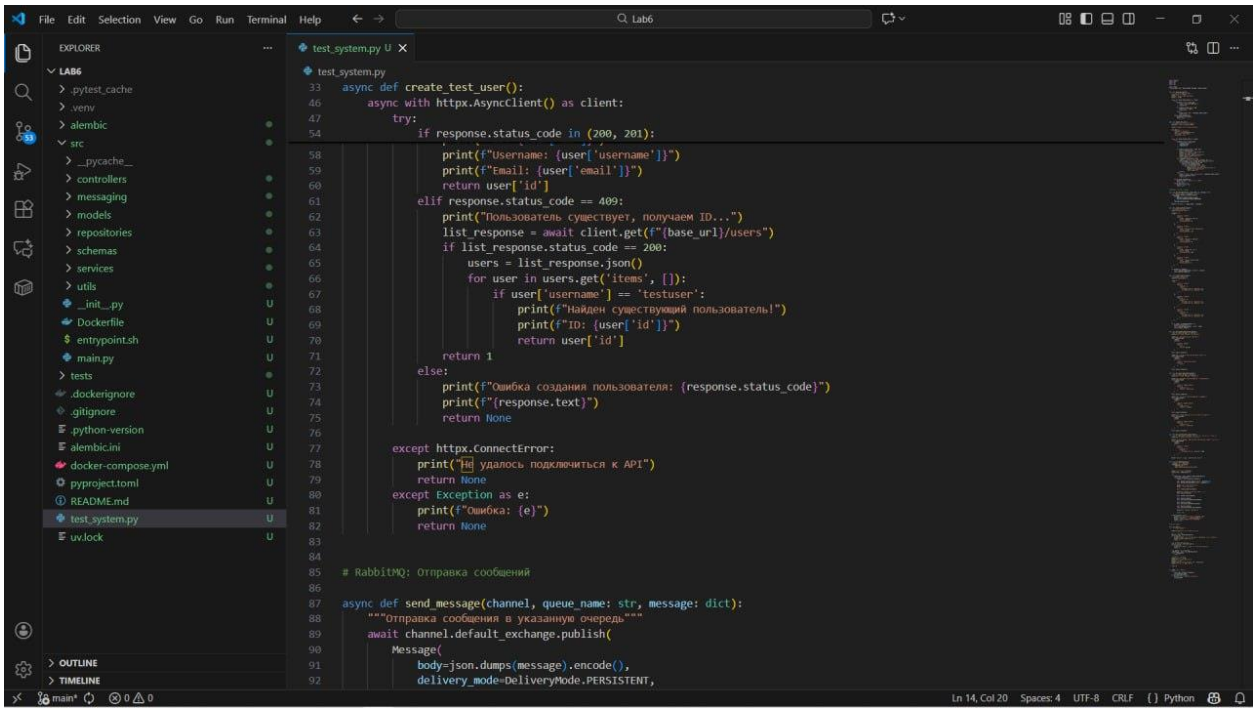
Файл: pyproject.toml - установлена библиотека aio-pika (выбрана асинхронная версия).

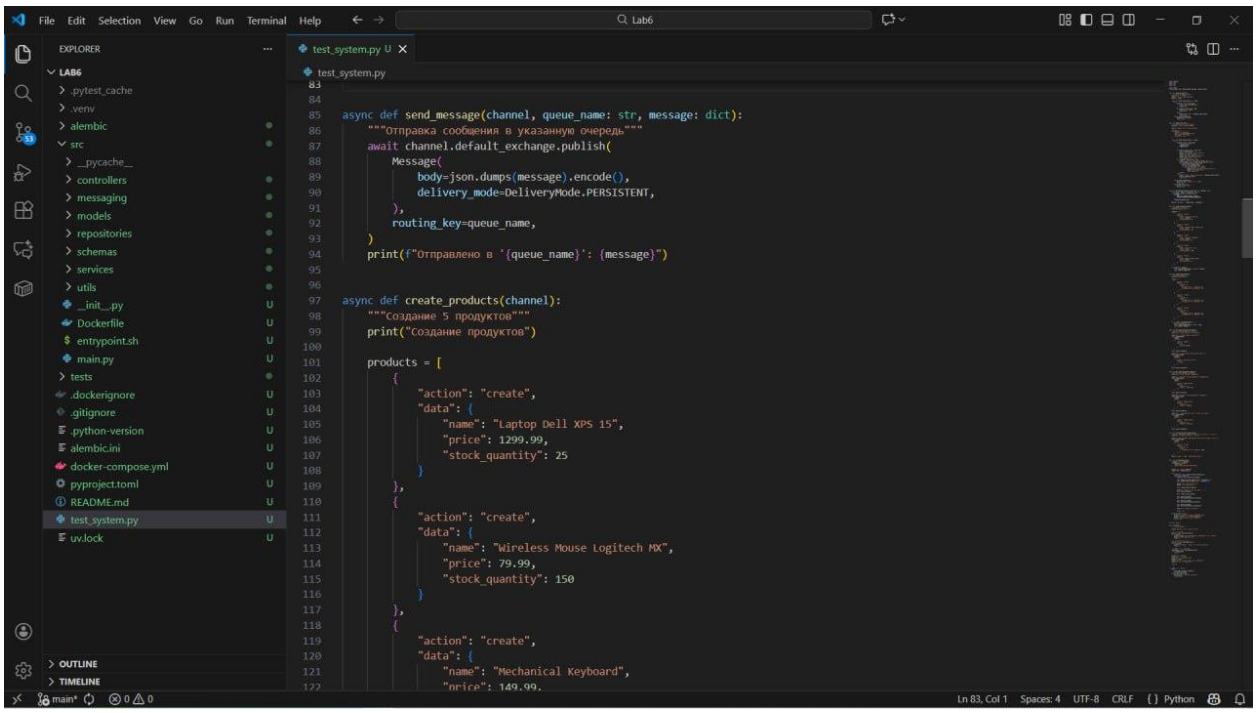
Файл: *test_system.py*

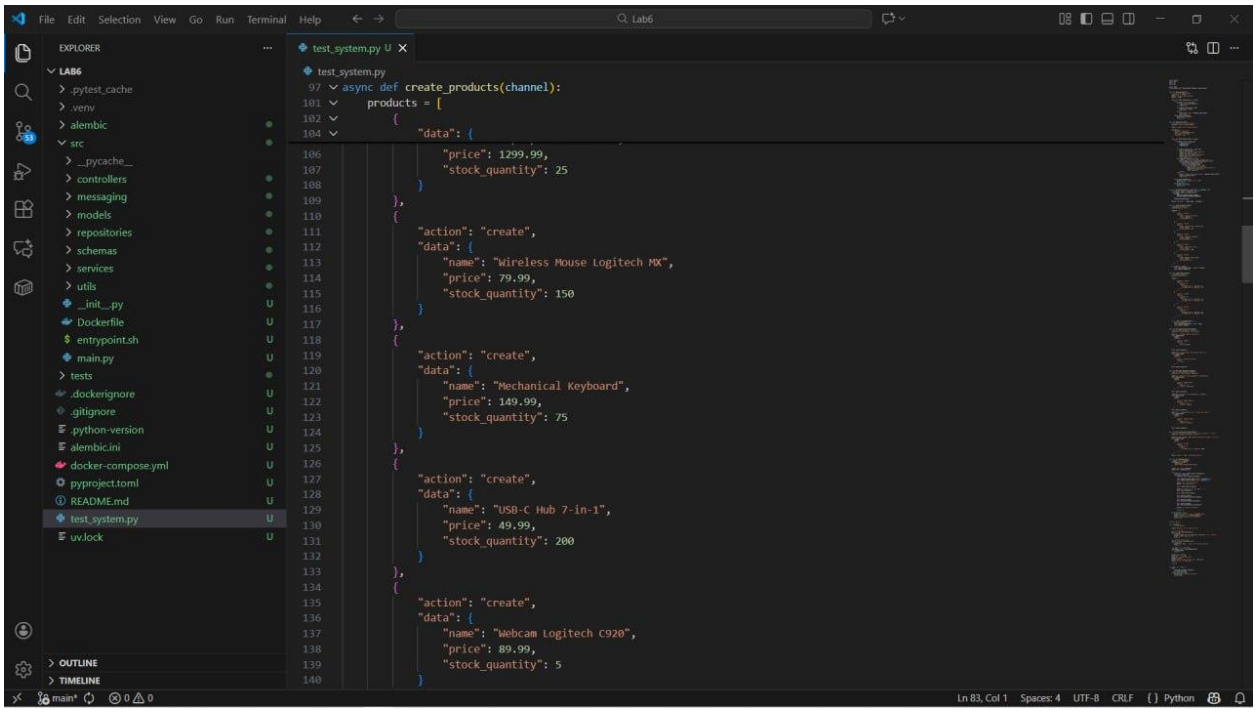
(скрипт продюсера)

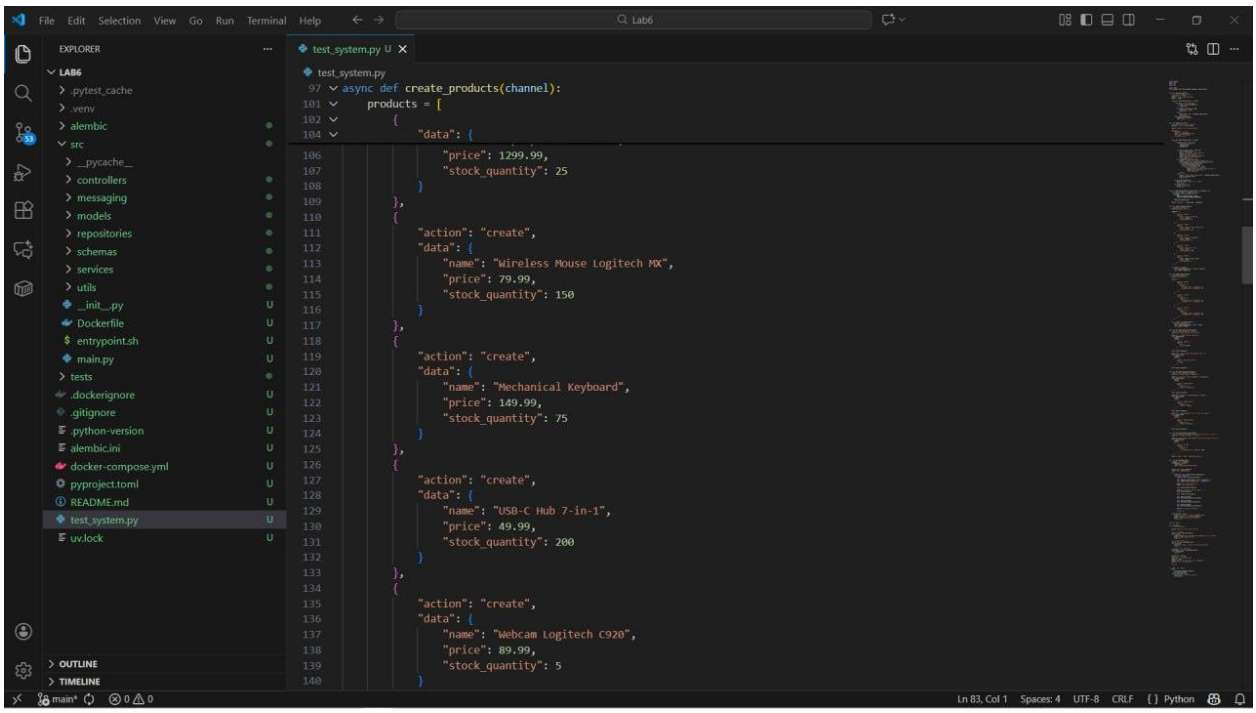


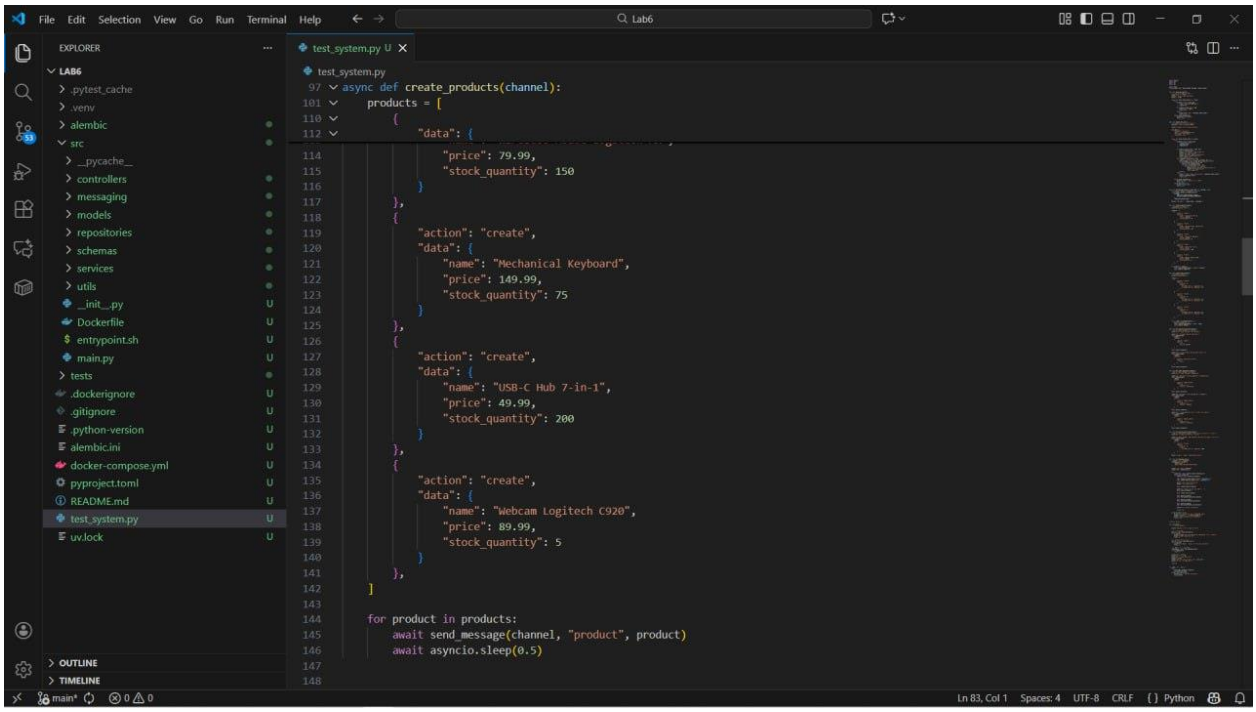

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def check_api_health():
async def create_test_user():
"""Создание тестового пользователя"""
base_url = "http://localhost:8000"
print("Создание тестового пользователя")
user_data = {
"username": "testuser",
"email": "testuser@example.com",
"full_name": "Test User",
"is_active": True
}
async with httpx.AsyncClient() as client:
try:
response = await client.post(
f"{base_url}/users",
json=user_data,
timeout=10.0
)
if response.status_code in (200, 201):
user = response.json()
print(f"Пользователь создан успешно!")
print(f"ID: {user['id']}")
print(f"Username: {user['username']}")
print(f"Email: {user['email']}")
return user['id']
elif response.status_code == 409:
print("Пользователь существует, получаем ID...")
list_response = await client.get(f"{base_url}/users")
if list_response.status_code == 200:
users = list_response.json()
for user in users.get('items', []):
if user['username'] == 'testuser':
print(f"Найден существующий пользователь!")
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 14, Col 20
Spaces: 4
UTF-8
CRLF
{} Python

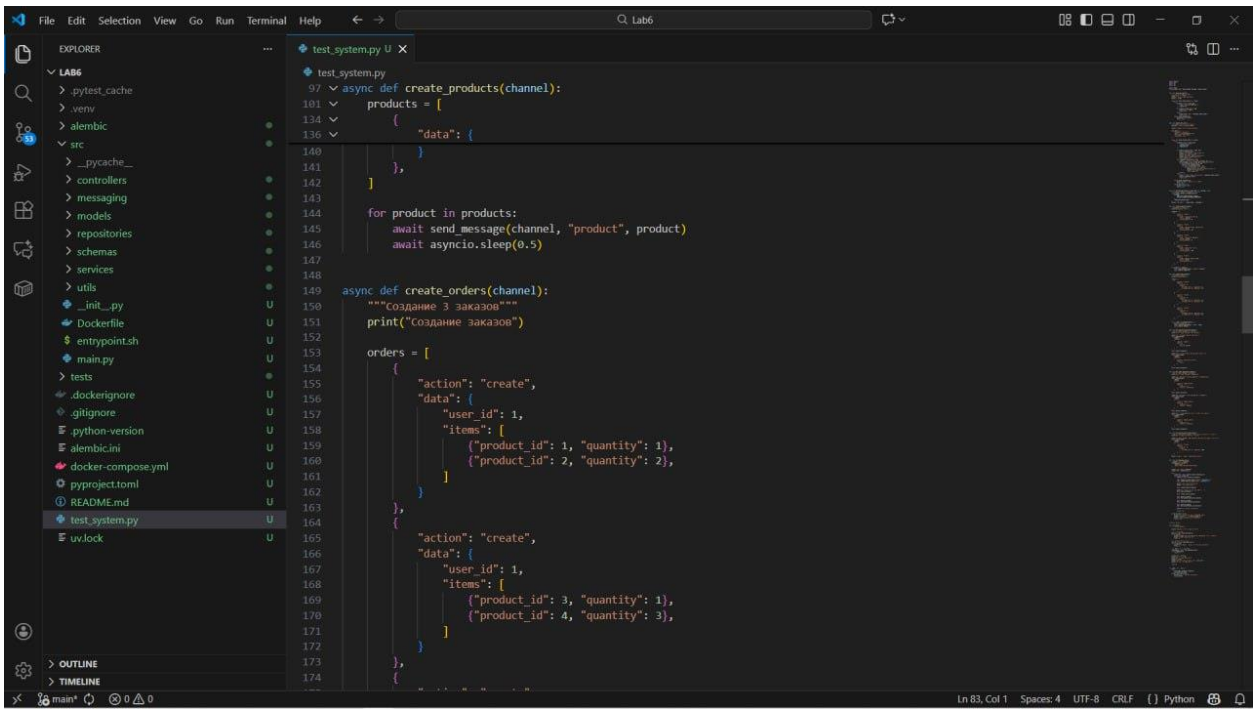

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def create_test_user():
async with httpx.AsyncClient() as client:
try:
if response.status_code in (200, 201):
print(f"Username: {user['username']}")
print(f"Email: {user['email']}")
return user['id']
elif response.status_code == 409:
print("Пользователь существует, получаем ID...")
list_response = await client.get(f"{base_url}/users")
if list_response.status_code == 200:
users = list_response.json()
for user in users.get('items', []):
if user['username'] == 'testuser':
print(f"Найден существующий пользователь!")
print(f"ID: {user['id']}")
return user['id']
return 1
else:
print(f"Ошибка создания пользователя: {response.status_code}")
print(f"{response.text}")
return None
except httpx.ConnectError:
print("Не удалось подключиться к API")
return None
except Exception as e:
print(f"Ошибка: {e}")
return None
RabbitMQ: Отправка сообщений
async def send_message(channel, queue_name: str, message: dict):
"""Отправка сообщения в указанную очередь"""
await channel.default_exchange.publish(
Message(
body=json.dumps(message).encode(),
delivery_mode=DeliveryMode.PERSISTENT,
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 14, Col 20
Spaces: 4
UTF-8
CRLF
{} Python

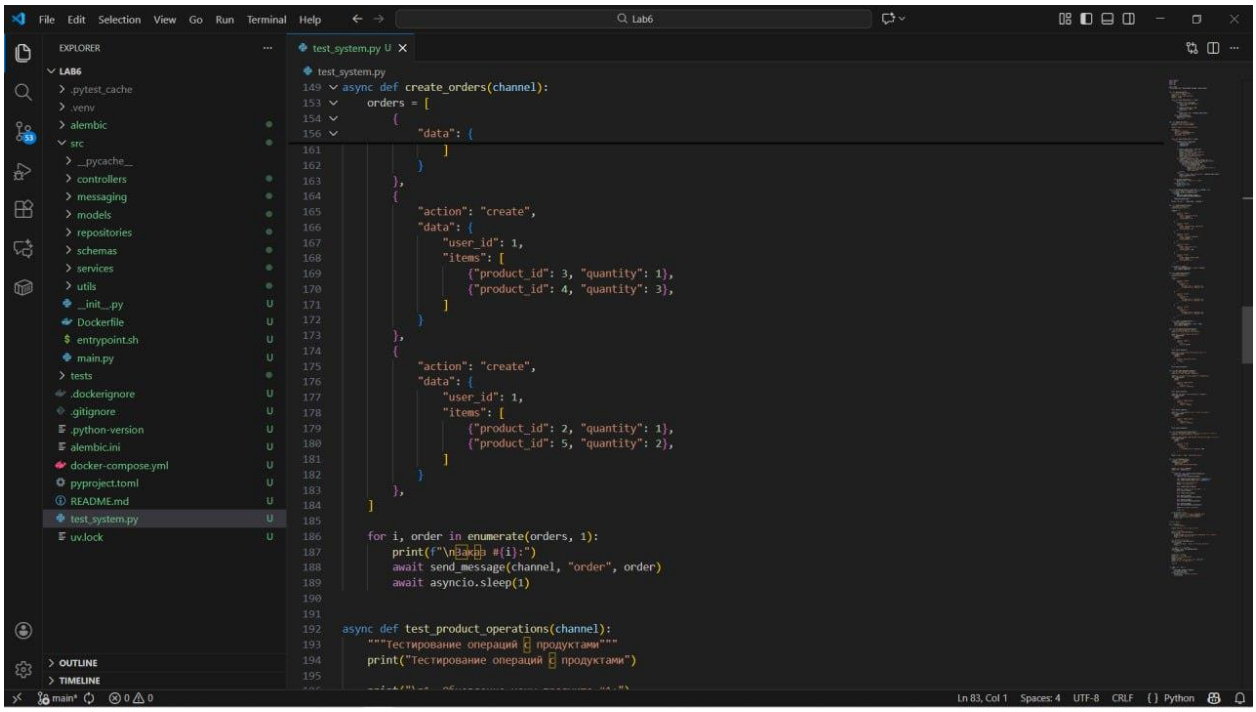

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def send_message(channel, queue_name: str, message: dict):
"""Отправка сообщения в указанную очередь"""
await channel.default_exchange.publish(
Message(
body=json.dumps(message).encode(),
delivery_mode=DeliveryMode.PERSISTENT,
),
routing_key=queue_name,
)
print(f"Отправлено в '{queue_name}': {message}")
async def create_products(channel):
"""Создание 5 продуктов"""
print("Создание продуктов")
products = [
{
"action": "create",
"data": {
"name": "Laptop Dell XPS 15",
"price": 1299.99,
"stock_quantity": 25
}
},
{
"action": "create",
"data": {
"name": "Wireless Mouse Logitech MX",
"price": 79.99,
"stock_quantity": 150
}
},
{
"action": "create",
"data": {
"name": "Mechanical Keyboard",
"price": 149.99,
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
Python

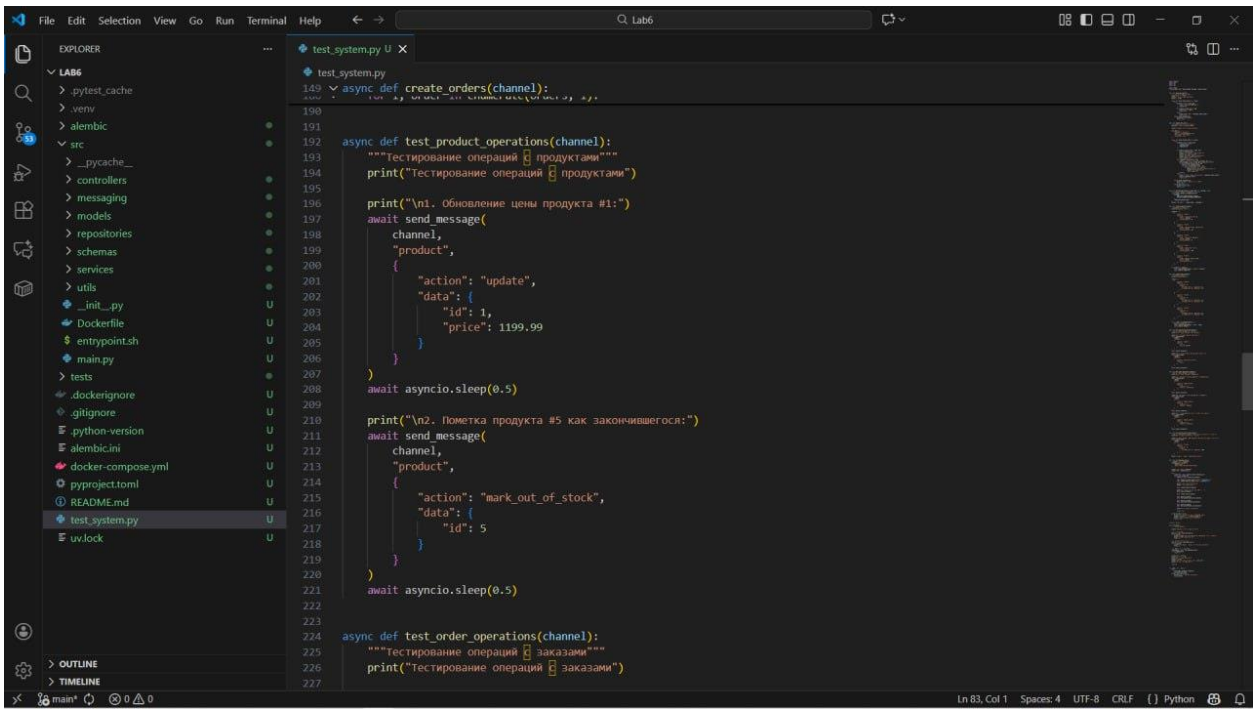

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def create_products(channel):
products = [
{
"data": {
"price": 1299.99,
"stock_quantity": 25
}
},
{
"action": "create",
"data": {
"name": "Wireless Mouse Logitech MX",
"price": 79.99,
"stock_quantity": 150
}
},
{
"action": "create",
"data": {
"name": "Mechanical Keyboard",
"price": 149.99,
"stock_quantity": 75
}
},
{
"action": "create",
"data": {
"name": "USB-C Hub 7-in-1",
"price": 49.99,
"stock_quantity": 200
}
},
{
"action": "create",
"data": {
"name": "Webcam Logitech C920",
"price": 89.99,
"stock_quantity": 5
}
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
Python

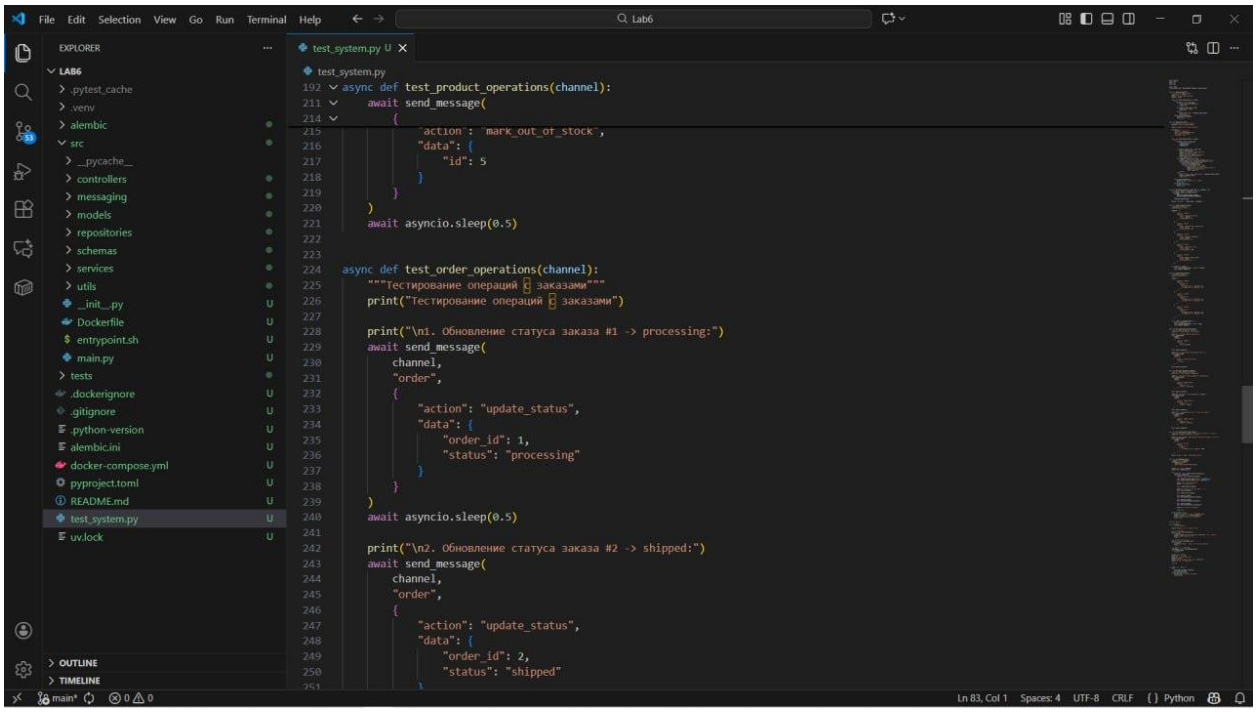

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def create_products(channel):
products = [
{
"data": {
"price": 1299.99,
"stock_quantity": 25
}
},
{
"action": "create",
"data": {
"name": "Wireless Mouse Logitech MX",
"price": 79.99,
"stock_quantity": 150
}
},
{
"action": "create",
"data": {
"name": "Mechanical Keyboard",
"price": 149.99,
"stock_quantity": 75
}
},
{
"action": "create",
"data": {
"name": "USB-C Hub 7-in-1",
"price": 49.99,
"stock_quantity": 200
}
},
{
"action": "create",
"data": {
"name": "Webcam Logitech C920",
"price": 89.99,
"stock_quantity": 5
}
OUTLINE
TIMELINE
Ln 83, Col 1 Spaces: 4 UTF-8 CRLF { } Python

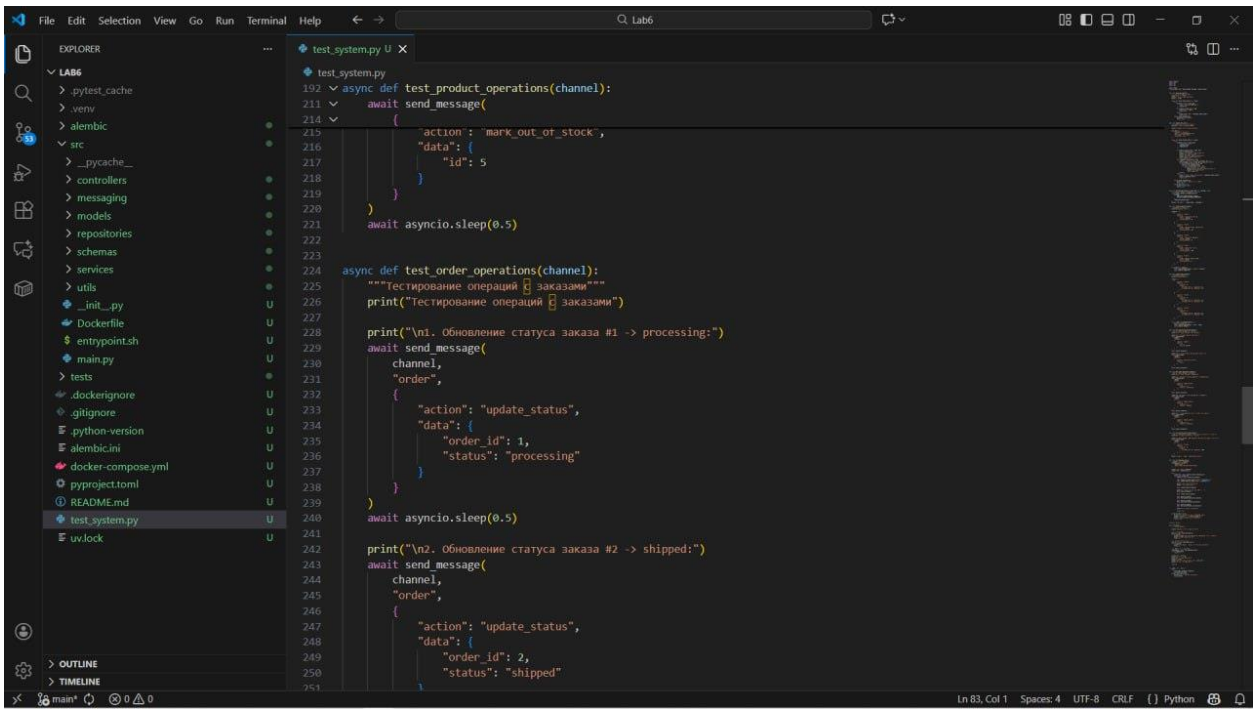

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def create_products(channel):
products = [
{
"data": {
"price": 79.99,
"stock_quantity": 150
}
},
{
"action": "create",
"data": {
"name": "Mechanical Keyboard",
"price": 149.99,
"stock_quantity": 75
}
},
{
"action": "create",
"data": {
"name": "USB-C Hub 7-in-1",
"price": 49.99,
"stock_quantity": 200
}
},
{
"action": "create",
"data": {
"name": "Webcam Logitech C920",
"price": 89.99,
"stock_quantity": 5
}
},
]
for product in products:
await send_message(channel, "product", product)
await asyncio.sleep(0.5)
OUTLINE
TIMELINE
Ln 83, Col 1 Spaces: 4 UTF-8 CRLF { } Python

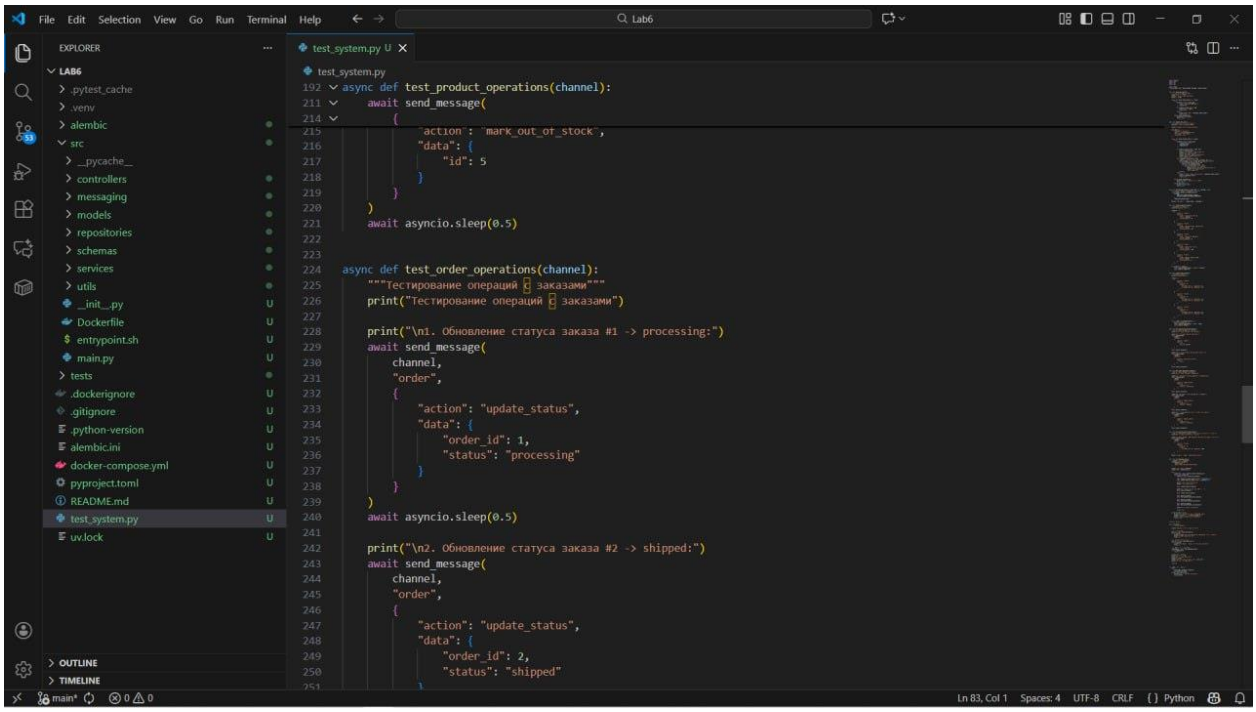

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def create_products(channel):
products = [
"data": {
}
},
]
for product in products:
await send_message(channel, "product", product)
await asyncio.sleep(0.5)
async def create_orders(channel):
"""Создание 3 заказов"""
print("Создание заказов")
orders = [
{
"action": "create",
"data": {
"user_id": 1,
"items": [
{"product_id": 1, "quantity": 1},
{"product_id": 2, "quantity": 2},
]
}
},
{
"action": "create",
"data": {
"user_id": 1,
"items": [
{"product_id": 3, "quantity": 1},
{"product_id": 4, "quantity": 3},
]
}
},
{
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
{} Python

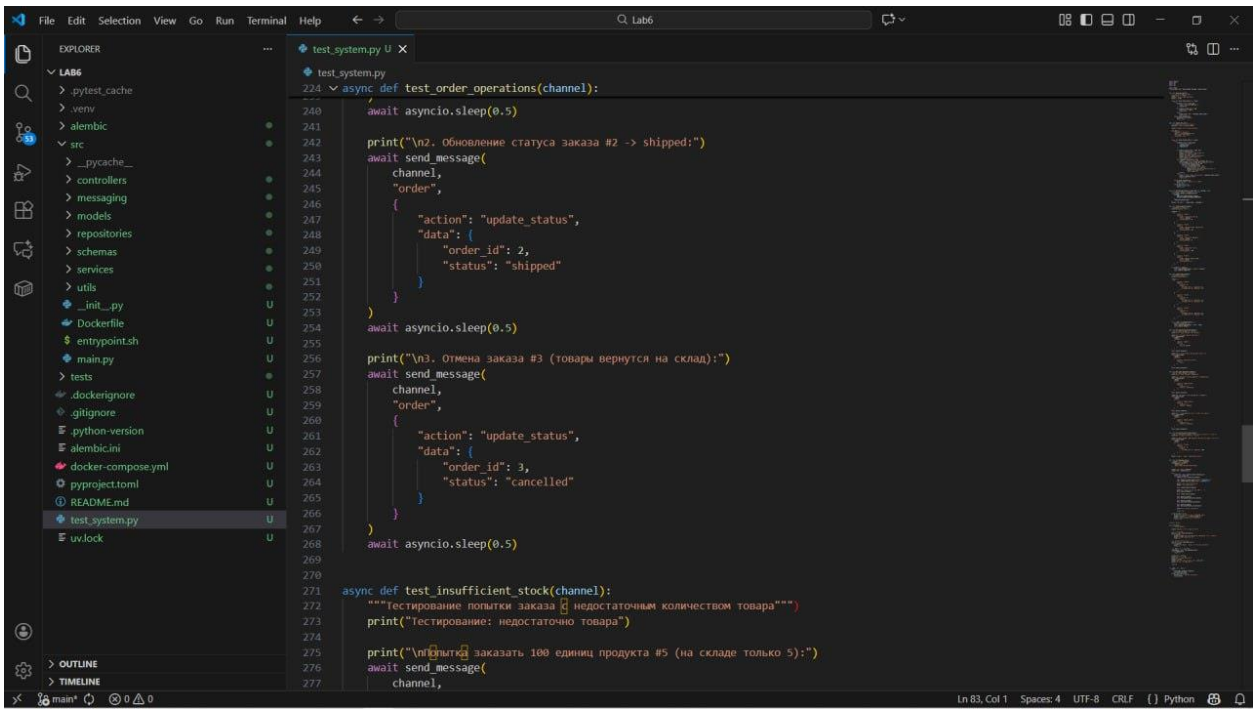

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def create_orders(channel):
orders = [
{
"data": {
]
}
},
{
"action": "create",
"data": {
"user_id": 1,
"items": [
{"product_id": 3, "quantity": 1},
{"product_id": 4, "quantity": 3},
]
}
},
{
"action": "create",
"data": {
"user_id": 1,
"items": [
{"product_id": 2, "quantity": 1},
{"product_id": 5, "quantity": 2},
]
}
},
]
for i, order in enumerate(orders, 1):
print(f"\nЗаказ #{i}:")
await send_message(channel, "order", order)
await asyncio.sleep(1)
async def test_product_operations(channel):
"""Тестирование операций с продуктами"""
print("Тестирование операций с продуктами")
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
{} Python

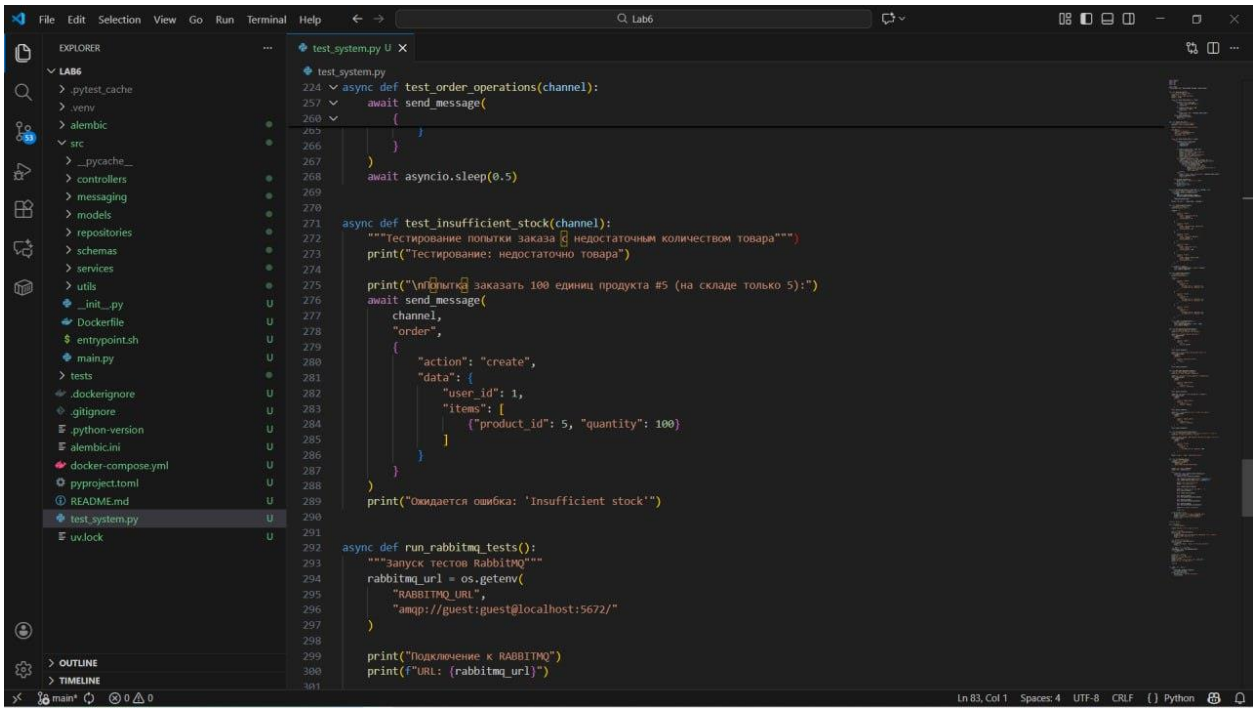

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def create_orders(channel):
async def test_product_operations(channel):
"""Тестирование операций с продуктами"""
print("Тестирование операций с продуктами")
print("\n1. Обновление цены продукта #1:")
await send_message(
channel,
"product",
{
"action": "update",
"data": {
"id": 1,
"price": 1199.99
}
}
)
await asyncio.sleep(0.5)
print("\n2. Пометка продукта #5 как закончившегося:")
await send_message(
channel,
"product",
{
"action": "mark_out_of_stock",
"data": {
"id": 5
}
}
)
await asyncio.sleep(0.5)
async def test_order_operations(channel):
"""Тестирование операций с заказами"""
print("Тестирование операций с заказами")
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
{ } Python

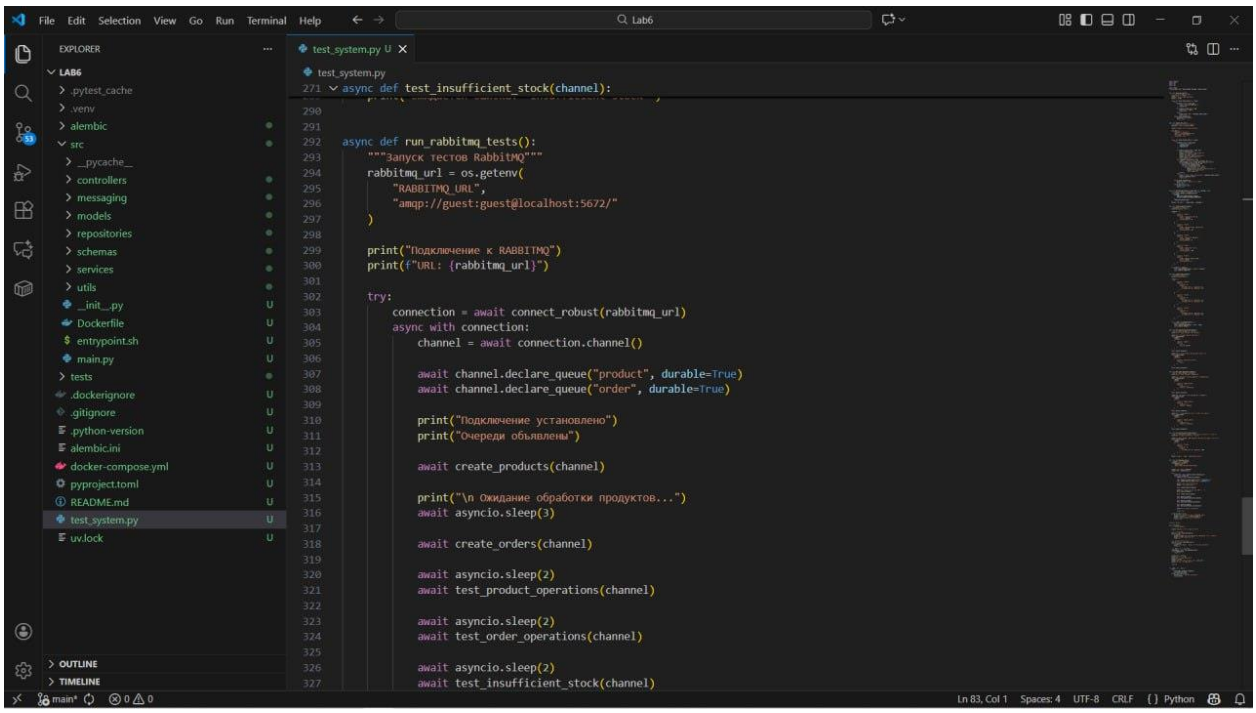

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def test_product_operations(channel):
await send_message(
{
"action": "mark_out_of_stock",
"data": {
"id": 5
}
}
)
await asyncio.sleep(0.5)
async def test_order_operations(channel):
"""Тестирование операций с заказами"""
print("Тестирование операций с заказами")
print("\n1. Обновление статуса заказа #1 -> processing:")
await send_message(
channel,
"order",
{
"action": "update_status",
"data": {
"order_id": 1,
"status": "processing"
}
}
)
await asyncio.sleep(0.5)
print("\n2. Обновление статуса заказа #2 -> shipped:")
await send_message(
channel,
"order",
{
"action": "update_status",
"data": {
"order_id": 2,
"status": "shipped"
}
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
{ } Python

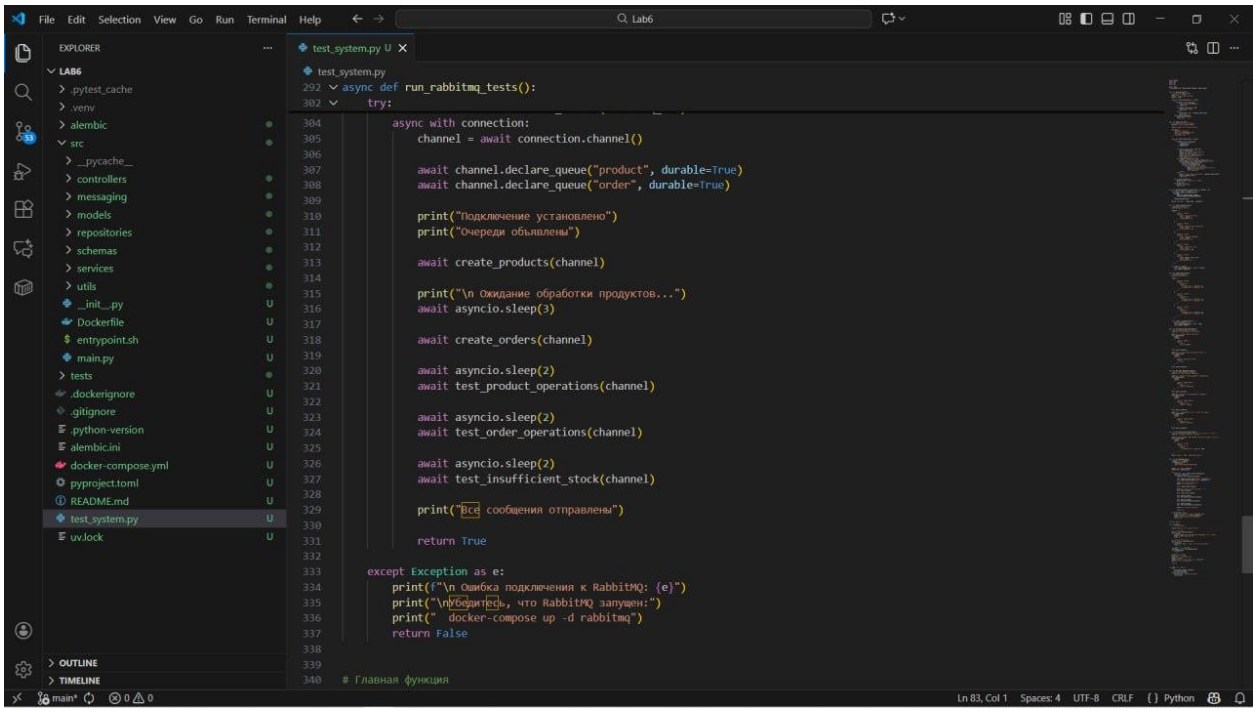

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def test_product_operations(channel):
await send_message(
"action": "mark_out_of_stock",
"data": {
"id": 5
}
)
await asyncio.sleep(0.5)
async def test_order_operations(channel):
"""Тестирование операций с заказами"""
print("Тестирование операций с заказами")
print("\n1. Обновление статуса заказа #1 -> processing:")
await send_message(
channel,
"order",
{
"action": "update_status",
"data": {
"order_id": 1,
"status": "processing"
}
}
)
await asyncio.sleep(0.5)
print("\n2. Обновление статуса заказа #2 -> shipped:")
await send_message(
channel,
"order",
{
"action": "update_status",
"data": {
"order_id": 2,
"status": "shipped"
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
Python

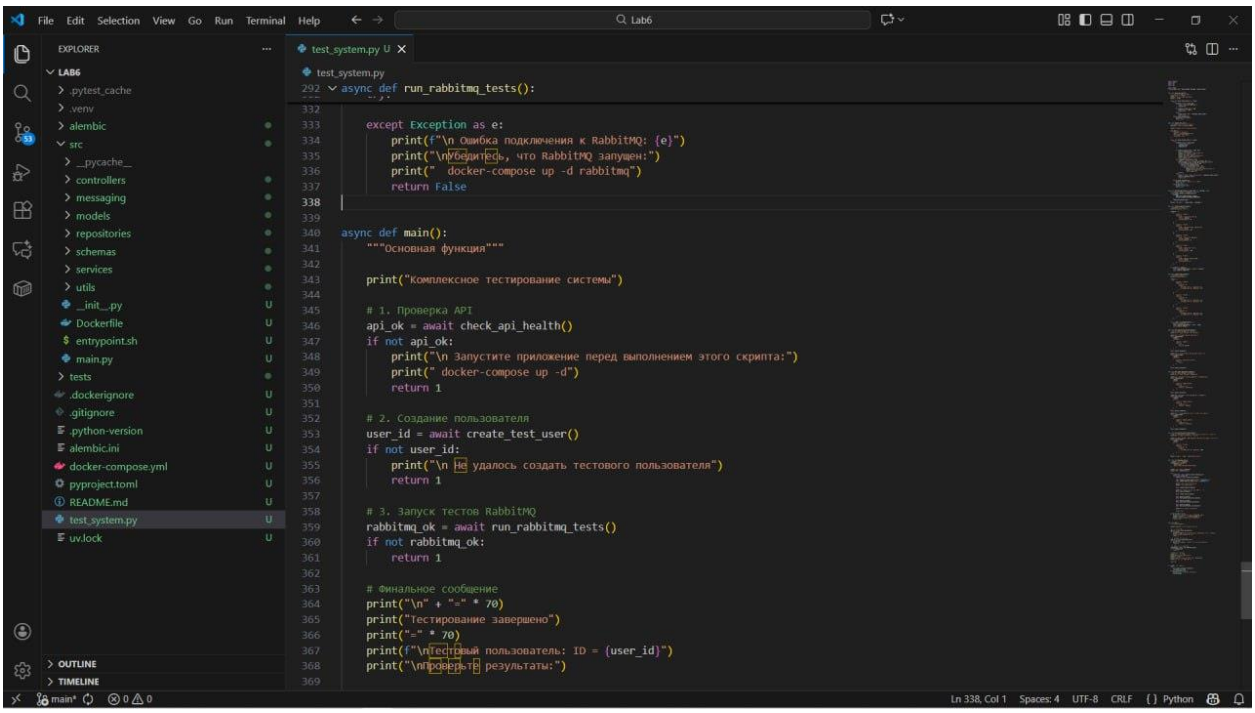

File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def test_product_operations(channel):
await send_message(
"action": "mark_out_of_stock",
"data": {
"id": 5
}
)
await asyncio.sleep(0.5)
async def test_order_operations(channel):
"""Тестирование операций с заказами"""
print("Тестирование операций с заказами")
print("\n1. Обновление статуса заказа #1 -> processing:")
await send_message(
channel,
"order",
{
"action": "update_status",
"data": {
"order_id": 1,
"status": "processing"
}
}
)
await asyncio.sleep(0.5)
print("\n2. Обновление статуса заказа #2 -> shipped:")
await send_message(
channel,
"order",
{
"action": "update_status",
"data": {
"order_id": 2,
"status": "shipped"
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
Python


File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
test_system.py U
LAB6
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
async def test_order_operations(channel):
await asyncio.sleep(0.5)
print("\n2. Обновление статуса заказа #2 -> shipped:")
await send_message(
channel,
"order",
{
"action": "update_status",
"data": {
"order_id": 2,
"status": "shipped"
}
}
)
await asyncio.sleep(0.5)
print("\n3. Отмена заказа #3 (товары вернутся на склад):")
await send_message(
channel,
"order",
{
"action": "update_status",
"data": {
"order_id": 3,
"status": "cancelled"
}
}
)
await asyncio.sleep(0.5)
async def test_insufficient_stock(channel):
"""Тестирование попытки заказа с недостаточным количеством товара"""
print("Тестирование: недостаточно товара")
print("\nПопытка заказать 100 единиц продукта #5 (на складе только 5):")
await send_message(
channel,
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
Python


File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
test_system.py U
LAB6
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
async def test_order_operations(channel):
await send_message(
{
}
}
)
await asyncio.sleep(0.5)
async def test_insufficient_stock(channel):
"""Тестирование попытки заказа с недостаточным количеством товара"""
print("Тестирование: недостаточно товара")
print("\nПопытка заказать 100 единиц продукта #5 (на складе только 5):")
await send_message(
channel,
"order",
{
"action": "create",
"data": {
"user_id": 1,
"items": [
{"product_id": 5, "quantity": 100}
]
}
}
)
print("Ожидается ошибка: 'Insufficient stock'")
async def run_rabbitmq_tests():
"""Запуск тестов RabbitMQ"""
rabbitmq_url = os.getenv(
"RABBITMQ_URL",
"amqp://guest:guest@localhost:5672/"
)
print("Подключение к RABBITMQ")
print(f"URL: {rabbitmq_url}")
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
Python


File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def test_insufficient_stock(channel):
async def run_rabbitmq_tests():
"""Запуск тестов RabbitMQ"""
rabbitmq_url = os.getenv(
"RABBITMQ_URL",
"amqp://guest:guest@localhost:5672/"
)
print("Подключение к RABBITMQ")
print(f"URL: {rabbitmq_url}")
try:
connection = await connect_robust(rabbitmq_url)
async with connection:
channel = await connection.channel()
await channel.declare_queue("product", durable=True)
await channel.declare_queue("order", durable=True)
print("Подключение установлено")
print("Очереди объявлены")
await create_products(channel)
print("\n Ожидание обработки продуктов...")
await asyncio.sleep(3)
await create_orders(channel)
await asyncio.sleep(2)
await test_product_operations(channel)
await asyncio.sleep(2)
await test_order_operations(channel)
await asyncio.sleep(2)
await test_insufficient_stock(channel)
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
Python


File Edit Selection View Go Run Terminal Help
Lab6
EXPLORER
LAB6
test_system.py U
test_system.py
async def run_rabbitmq_tests():
try:
async with connection:
channel = await connection.channel()
await channel.declare_queue("product", durable=True)
await channel.declare_queue("order", durable=True)
print("Подключение установлено")
print("Очереди объявлены")
await create_products(channel)
print("\n Ожидание обработки продуктов...")
await asyncio.sleep(3)
await create_orders(channel)
await asyncio.sleep(2)
await test_product_operations(channel)
await asyncio.sleep(2)
await test_order_operations(channel)
await asyncio.sleep(2)
await test_insufficient_stock(channel)
print("Все сообщения отправлены")
return True
except Exception as e:
print(f"\n Ошибка подключения к RabbitMQ: {e}")
print("\nУбедитесь, что RabbitMQ запущен:")
print(" docker-compose up -d rabbitmq")
return False
Главная функция
.pytest_cache
.venv
alembic
src
__pycache__
controllers
messaging
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
test_system.py
uv.lock
OUTLINE
TIMELINE
main*
Ln 83, Col 1
Spaces: 4
UTF-8
CRLF
Python



Данный скрипт представляет собой комплексный тестовый клиент для проверки работы системы, состоящей из REST API (FastAPI) и асинхронного брокера сообщений RabbitMQ.

Краткое описание функциональности:

Проверка доступности (check_api_health):

- Отправляет HTTP-запрос к эндпоинту /users для проверки работоспособности основного API.

Подготовка тестовых данных (create_test_user):

- Создает в системе тестового пользователя через API. Если пользователь уже существует — получает его ID.

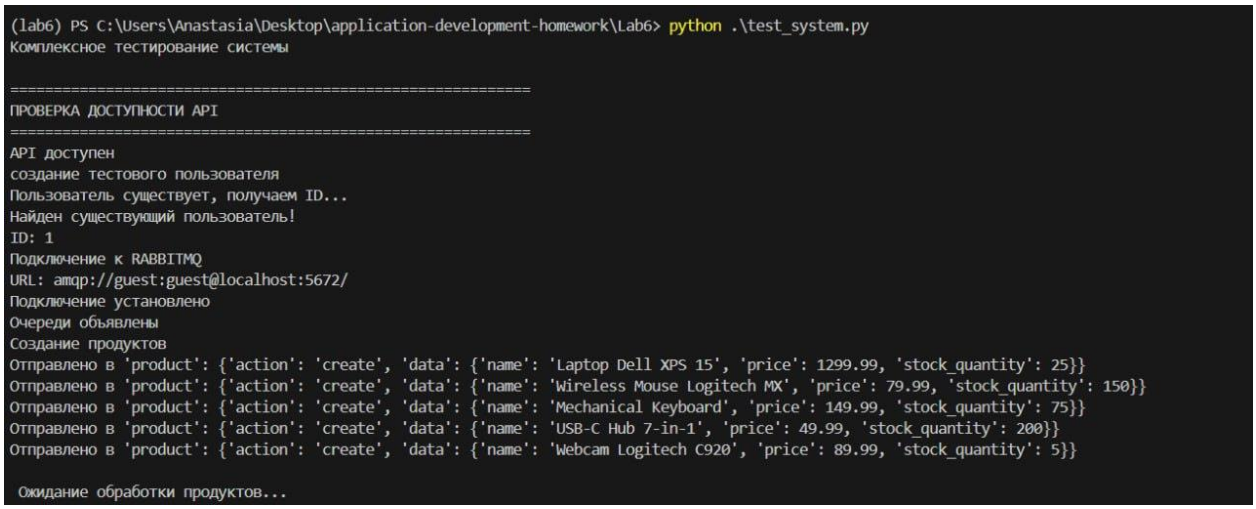
Интеграционное тестирование через RabbitMQ (run_rabbitmq_tests):

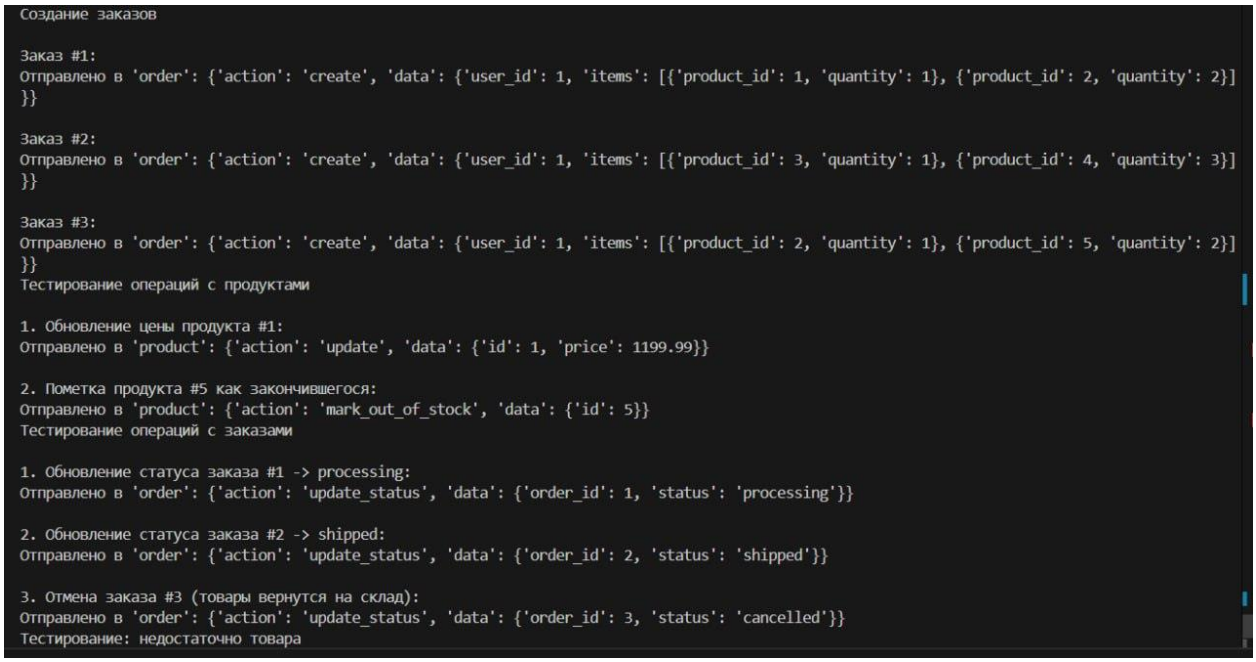
- Устанавливает соединение с RabbitMQ и объявляет две очереди: product и order.

Сценарий тестирования:

- Создание товаров: отправляет в очередь product 5 сообщений для создания товаров.
- Создание заказов: отправляет в очередь order 3 сообщения для создания заказов от имени тестового пользователя.

- Тест операций с товарами: имитирует обновление цены товара и пометку его как отсутствующего на складе.
- Тест операций с заказами: имитирует изменение статусов заказов (в обработке, отправлен, отменен).
- Тест обработки ошибок: пытается создать заказ с количеством товара, превышающим остаток на складе, для проверки корректной валидации





Результат: 5 продуктов + 3 заказа отправлены в очереди RabbitMQ

Общий поток данных

Асинхронный поток (через RabbitMQ)

RabbitMQ
→ FastStream
→ Queue Handler
→ Service
→ Repository
→ PostgreSQL

Синхронный поток (через HTTP)

HTTP Client
→ Litestar Controller
→ Service
→ Repository
→ PostgreSQL

Бизнес-логика обработки заказов

Сценарий 1: Успешное создание заказа

- Приходит сообщение в очередь order с action create
- Проверяется существование пользователя
- Проверяется существование всех продуктов
- Проверяется наличие товара на складе для каждой позиции

Если всё хорошо:

- Создается заказ со статусом pending
- Товары списываются со склада
- Рассчитывается общая сумма заказа

Сценарий 2: Недостаточно товара на складе

- Приходит заказ с quantity: 10 для продукта, у которого stock_quantity: 5
- Выбрасывается ошибка "Insufficient stock"
- Заказ НЕ создается
- Товары НЕ списываются

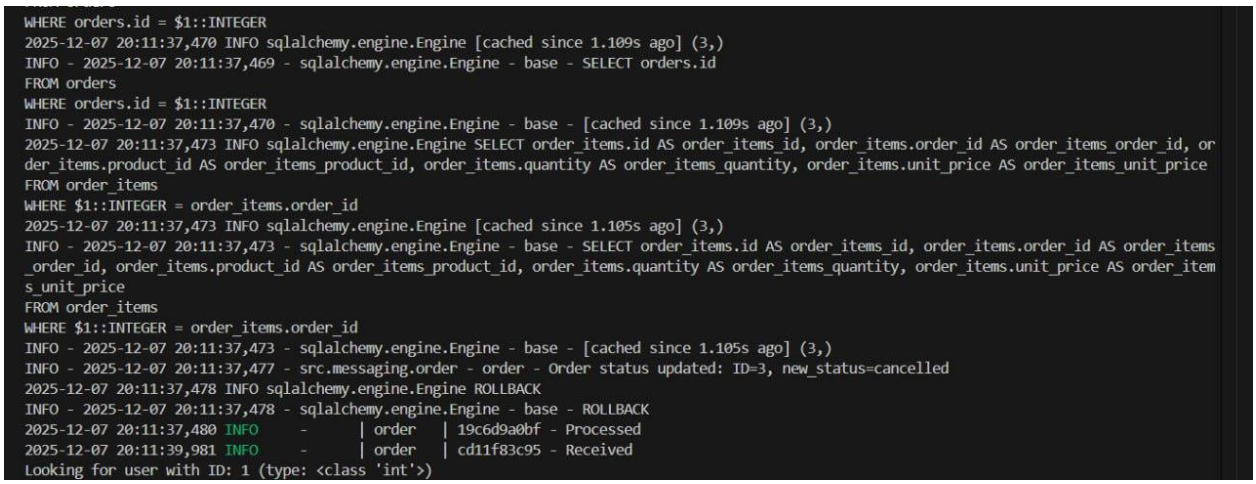
Сценарий 3: Отмена заказа

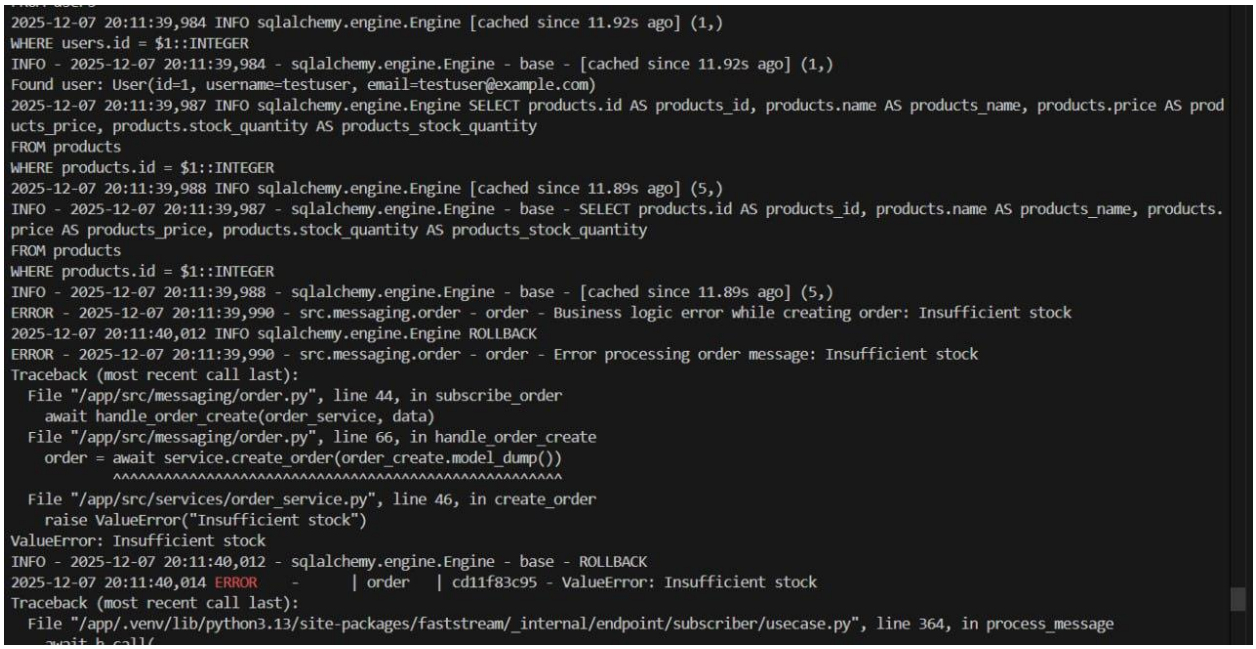
- Приходит сообщение с action update_status и status: "cancelled"
- Находится заказ
- Для каждой позиции заказа товары возвращаются на склад
- Статус заказа меняется на cancelled

Сценарий 4: Продукт закончился на складе

- Администратор отправляет сообщение с action mark_out_of_stock
- У продукта устанавливается stock_quantity: 0
- Все последующие заказы с этим продуктом будут отклонены с ошибкой "Insufficient stock"

Пример логов приложения





Состояние базы после тестирования

Server response

Code	Details
200	Response body


```
{
  "total": 1,
  "items": [
    {
      "username": "testuser",
      "email": "testuser@example.com",
      "full_name": "Test User",
      "is_active": true,
      "id": 1,
      "created_at": "2025-12-07T20:19:51.141710Z",
      "updated_at": "2025-12-07T20:19:51.141710Z"
    }
  ]
}
```


Response headers

Code	Details
200	Response body


```
{
  "total": 5,
  "items": [
    {
      "id": 3,
      "name": "Mechanical Keyboard",
      "price": 149.99,
      "stock_quantity": 74
    },
    {
      "id": 4,
      "name": "USB-C Hub 7-in-1",
      "price": 49.99,
      "stock_quantity": 197
    },
    {
      "id": 1,
      "name": "Laptop Dell XPS 15",
      "price": 1199.99,
      "stock_quantity": 24
    },
    {
      "id": 2,
      "name": "Wireless Mouse Logitech MX",
      "price": 79.99,
      "stock_quantity": 148
    },
```


Response headers

Code	Details
200	Response body


```
    },
    {
      "id": 4,
      "name": "USB-C Hub 7-in-1",
      "price": 49.99,
      "stock_quantity": 197
    },
    {
      "id": 1,
      "name": "Laptop Dell XPS 15",
      "price": 1199.99,
      "stock_quantity": 24
    },
    {
      "id": 2,
      "name": "Wireless Mouse Logitech MX",
      "price": 79.99,
      "stock_quantity": 148
    },
    {
      "id": 5,
      "name": "Webcam Logitech C920",
      "price": 89.99,
      "stock_quantity": 2
    }
  ]
}
```


Code	Details
200	Response body


```
{
  "total": 3,
  "items": [
    {
      "id": 1,
      "user_id": 1,
      "total_amount": 1459.97,
      "status": "processing",
      "items": [
        {
          "id": 1,
          "product_id": 1,
          "quantity": 1,
          "unit_price": 1299.99
        },
        {
          "id": 2,
          "product_id": 2,
          "quantity": 2,
          "unit_price": 79.99
        }
      ]
    },
    {
      "id": 2,
      "user_id": 1,
      "total_amount": 299.96000000000004,
```

Response body


```
      }
    ]
  },
  {
    "id": 2,
    "user_id": 1,
    "total_amount": 299.96000000000004,
    "status": "shipped",
    "items": [
      {
        "id": 3,
        "product_id": 3,
        "quantity": 1,
        "unit_price": 149.99
      },
      {
        "id": 4,
        "product_id": 4,
        "quantity": 3,
        "unit_price": 49.99
      }
    ]
  },
  {
    "id": 3,
    "user_id": 1,
    "total_amount": 259.96999999999997,
```


Response headers

200

Response body


```
          "quantity": 3,
          "unit_price": 49.99
        }
      ]
    },
    {
      "id": 3,
      "user_id": 1,
      "total_amount": 259.96999999999997,
      "status": "cancelled",
      "items": [
        {
          "id": 5,
          "product_id": 2,
          "quantity": 1,
          "unit_price": 79.99
        },
        {
          "id": 6,
          "product_id": 5,
          "quantity": 2,
          "unit_price": 89.99
        }
      ]
    }
  ]
}
```

Ответы на вопросы

1. Что такое AMQP? Каковы его основные преимущества?

AMQP (Advanced Message Queuing Protocol) - открытый протокол для обмена сообщениями между приложениями через брокер сообщений (например RabbitMQ). Он определяет формат сообщений, семантику подтверждений, маршрутизацию и управление очередями.

Основные преимущества:

- Стандартизованный протокол — совместимость между разными реализациями/клиентами.
- Надежность: гарантированная доставка сообщений с подтверждением (acknowledgments)
- Маршрутизация: гибкая система exchange и binding для направления сообщений
- Безопасность: встроенная аутентификация и шифрование (SASL, TLS)
- Независимость от платформы: единый стандарт для разных языков и систем
- Транзакционность: поддержка транзакций для атомарных операций

В работе используется через URL: amqp://guest:guest@rabbitmq:5672/

2. В чем разница между очередями сообщений и шинами сообщений?

Очередь сообщений - механизм асинхронного взаимодействия между приложениями, при котором сообщения, отправленные производителем (producer), временно сохраняются в упорядоченной структуре и доставляются ровно одному получателю (consumer)

- Имеет point-to-point модель: одно сообщение - один получатель
- Сообщения хранятся до обработки
- Гарантия порядка обработки (FIFO) - сообщение, которое первым попало в очередь, будет первым обработано.

Пример: очереди order и product с одним consumer на сообщение (только один обработчик получит это сообщение):

- @broker.subscriber(RabbitQueue("order", durable=True))
  async def subscribe_order(message: Dict):

Шина сообщений - архитектурный подход к обмену сообщениями между системами, обеспечивающая обмен сообщениями между множеством сервисов, маршрутизацию, трансформации, публикацию/подписку и согласованные контракты.

- Имеет Pub/Sub модель: одно сообщение - множество подписчиков
- Broadcast доставка всем подписчикам
- Используются topic/fanout exchanges

Пример: уведомления о событиях (order.created - email service, analytics, warehouse)

3. Как обеспечить надежную доставку сообщений в RabbitMQ?

Ключевые практики (комбинируются для надёжности):

1) **Durable очереди** (они переживают перезапуск RabbitMQ). Нужно создавать очереди с durable=true. Пример из лабораторной: @broker.subscriber(RabbitQueue("order", durable=True))

2) **Persistent сообщения** (сохраняются на диск). Это требуется, чтобы сообщения пережили перезапуск брокера. Пример: delivery_mode=DeliveryMode.PERSISTENT.

3) **Acknowledgments** (автоматические в FastStream): при успешной обработке сообщение удаляется из очереди, а при исключении сообщение возвращается в очередь для повторной обработки.

4) **Publisher confirms** – позволяет включать подтверждения (publisher confirms) как способ узнать, что брокер принял сообщение на запись.

5) **Обработка отказов: reject / nack + DLQ.** При ошибках сообщение либо возвращается в очередь (requeue=true), либо перенаправляется в Dead Letter Queue для дальнейшего анализа.

4. Что произойдет с сообщением, если consumer упадет во время обработки?

Конкретное поведение в этом случае зависит от настроек.

Если consumer использует manual acknowledgements (auto_ack=false), есть несколько вариантов:

- при условии, что потребитель не успел отправить ack и его соединение/канал закрывается, RabbitMQ пометит сообщение как unacknowledged и переложит его обратно в очередь — оно будет повторно доставлено другому (или тому же при восстановлении) consumer. При повторной доставке будет установлен флаг redelivered=true. **Этот вариант используется в коде лабораторной**

- при условии, если потребитель успел отправить ack до падения - то сообщение считается доставленным и удаляется из очереди.

Если consumer использует auto-ack = true (no-ack):

- в этом случае сообщение считается доставленным сразу после отправки брокером; если consumer падает во время обработки, брокер не будет повторно отправлять сообщение - обработка потеряна.

Если consumer сделал nack/reject с requeue=false: сообщение либо отброшено, либо попадёт в DLX (если настроен dead-letter-exchange).

5. Как обеспечить сохранность сообщений при перезапуске RabbitMQ?

Чтобы сообщения пережили перезапуск брокера в работе были реализованы:

1. Durable очереди (объявлены при старте): очереди пересоздаются после перезапуска с сохранением настроек.
2. Persistent сообщения: pythondelivery_mode=DeliveryMode.PERSISTENT. Сообщения записываются на диск перед подтверждением.
3. Volume для данных RabbitMQ (docker-compose.yml): все данные

(очереди, сообщения, настройки) сохраняются между перезапусками контейнера.

Что произойдет при перезапуске:

- Очереди восстановятся из volume
- Необработанные persistent сообщения останутся в очередях
- Non-persistent сообщения будут потеряны

6. Что такое TTL (Time To Live) для сообщений и как его настроить?

TTL (Time To Live) — максимальное время, в течение которого сообщение считается «жизнеспособным» в очереди. По истечении TTL сообщение либо удаляется, либо отправляется в Dead-Letter Exchange (если настроен DLX).

Применение:

- Предотвращение накопления устаревших сообщений
- Автоматическая очистка временных данных
- Защита от переполнения очередей

Способы настройки:

- На уровне сообщения через свойство expiration. Значение задаётся в миллисекундах строкой. Если у очереди тоже задан x-message-ttl, то используется минимальное значение.
- На уровне очереди через аргумент x-message-ttl. Это TTL для всех сообщений в очереди и классический вариант для бизнес-очередей.
- Для Dead Letter Queue (DLQ) часто используют для: временного хранения ошибок или реализации повторных попыток (retry).

Вывод:

В ходе выполнения лабораторной работы была реализована полноценная система обработки заказов и продуктов с использованием RabbitMQ как брокера сообщений, с разделением команд (через MQ) и запросов (через REST

API), обеспечивающая надежность, масштабируемость и согласованность данных.

Практическая реализация:

- Настроена инфраструктура в Docker Compose с RabbitMQ, PostgreSQL и приложением
- Созданы две очереди (order и product) с поддержкой множественных операций через единый обработчик
- Реализована бизнес-логика с проверкой остатков товаров, автоматическим списанием при создании заказа и возвратом при отмене
- Разделена ответственность: команды (create/update) через RabbitMQ, запросы (get/list) через REST API

Разработан комплексный тестовый скрипт, создающий 5 продуктов и 3 заказа, демонстрирующий все основные операции системы, включая негативные сценарии с недостаточным количеством товара.

Полученный опыт применим для построения микросервисных архитектур, систем обработки событий и асинхронных workflow, где требуется надежная доставка сообщений и отказоустойчивость.