

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.

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 `aiorika` 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*

```
1 services:
2   db:
3     image: postgres:15
4     environment:
5       POSTGRES_USER: user
6       POSTGRES_PASSWORD: superpass
7       POSTGRES_DB: db
8     ports:
9       - "5432:5432"
10    volumes:
11      - postgres_data:/var/lib/postgresql/data/
12    networks:
13      - db_network
14    healthcheck:
15      test: ["CMD-SHELL", "pg_isready -U user -d db"]
16      interval: 10s
17      timeout: 5s
18      retries: 5
19
20   pgadmin:
21     image: dpage/pgadmin4
22     environment:
23       PGADMIN_DEFAULT_EMAIL: admin@example.com
24       PGADMIN_DEFAULT_PASSWORD: superpass
25     ports:
26       - "8080:80"
27     depends_on:
28       - db
29     networks:
30       - db_network
```

```
1 services:
31
32 rabbitmq:
33   image: rabbitmq:3-management
34   ports:
35     - "5672:5672"
36     - "15672:15672"
37   environment:
38     - RABBITMQ_DEFAULT_VHOST=/
39     - RABBITMQ_DEFAULT_USER=guest
40     - RABBITMQ_DEFAULT_PASS=guest
41   volumes:
42     - rabbitmq_data:/var/lib/rabbitmq
43   networks:
44     - db_network
45   healthcheck:
46     test: ["CMD", "rabbitmq-diagnostics", "ping"]
47     interval: 10s
48     timeout: 5s
49     retries: 5
50     start_period: 40s
51
52
53
54
55
56
57
58
59
60
61
62
63
```

```
1 services:
32 rabbitmq:
45   healthcheck:
48     timeout: 5s
49     retries: 5
50     start_period: 40s
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
```

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

RabbitMQ 3.13.7 Erlang 26.2.5.16

Refreshed 2025-12-08 01:10:33 Refresh every 5 seconds

Virtual host All Cluster rabbit@d9963ff22931 User guest Log out

Overview Connections Channels Exchanges **Queues and Streams** Admin

Queues

All queues (2)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Displaying 2 items, page size up to: 100

Overview						Messages			Message rates			
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver	get	ack	
/	order	classic	D Args	running	0	0	0					
/	product	classic	D Args	running	0	0	0					

➤ Add a new queue

HTTP API Documentation Tutorials New releases Commercial edition Commercial support Discussions Discord Plugins GitHub

Файл: *pyproject.toml*

```

1 [project]
2 name = "lab6"
3 version = "0.1.0"
4 description = "Add your description here"
5 readme = "README.md"
6 requires-python = ">=3.13"
7 dependencies = [
8     "aio-pika>=9.4.3",
9     "aiosqlite>=0.21.0",
10    "alembic>=1.17.2",
11    "asynccpg>=0.30.0",
12    "black>=25.11.0",
13    "fast-depends>=3.0.5",
14    "faststream[rabbit]>=0.1.4",
15    "isort>=7.0.0",
16    "litedstar[standard]>=2.18.0",
17    "polyfactory>=3.0.0",
18    "pre-commit>=4.5.0",
19    "pydantic[email]>=2.12.4",
20    "pylint>=4.0.3",
21    "pytest>=9.0.1",
22    "pytest-asyncio>=1.3.0",
23    "sniffio>=1.3.1",
24    "sqlalchemy[asyncio]>=2.0.44",
25    "uvicorn[standard]>=0.38.0",
26 ]
27

```

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

```

1 import asyncio
2 import pytest
3 from faststream.rabbit import RabbitBroker
4
5 RABBITMQ_URL = "amqp://guest:guest@localhost:5672/"
6
7
8 @pytest.fixture
9 async def broker():
10     """Fixture to create and cleanup broker"""
11     broker = RabbitBroker(RABBITMQ_URL)
12     await broker.start()
13     yield broker
14     await broker.stop()
15
16
17 @pytest.mark.asyncio
18 async def test_rabbitmq_connection(broker):
19     """Test basic RabbitMQ connection"""
20     assert broker._connection is not None
21
22
23 @pytest.mark.asyncio
24 async def test_rabbitmq_publish_subscribe(broker):
25     """Test publish and subscribe functionality"""
26     received_messages = []
27
28     @broker.subscriber("test_queue")
29     async def handle(msg):
30         received_messages.append(msg)
31
32     await broker.start()
33
34     test_message = "Hello from pytest!"
35     await broker.publish(test_message, "test_queue")
36
37     await asyncio.sleep(1)
38
39     assert len(received_messages) == 1
40     assert received_messages[0] == test_message

```


The image displays two screenshots of a Visual Studio Code editor. The top screenshot shows the source code of a file named `test_rabbitmq.py` located in the `tests` directory. The code includes four asynchronous unit tests: `test_rabbitmq_publish_subscribe`, `test_rabbitmq_multiple_messages`, `test_rabbitmq_service_running`, and `test_rabbitmq_connection`. The bottom screenshot shows the terminal output of running these tests using the command `pytest .\tests\test_rabbitmq.py -v`. The output indicates that all four tests passed successfully within 4.59 seconds.

```
File Edit Selection View Go Run Terminal Help
test_rabbitmq.py X
tests > test_rabbitmq.py
24 async def test_rabbitmq_publish_subscribe(broker):
39     assert len(received_messages) == 1
40     assert received_messages[0] == test_message
41
42
43 @pytest.mark.asyncio
44 async def test_rabbitmq_multiple_messages(broker):
45     """Test multiple messages"""
46     received_messages = []
47
48     @broker.subscriber("multi_queue")
49     async def handle(msg):
50         received_messages.append(msg)
51
52     await broker.start()
53
54     messages = ["msg1", "msg2", "msg3"]
55     for msg in messages:
56         await broker.publish(msg, "multi_queue")
57
58     await asyncio.sleep(2)
59
60     assert len(received_messages) == len(messages)
61     assert received_messages == messages
62
63
64 def test_rabbitmq_service_running():
65     """Test if RabbitMQ service is accessible"""
66     import socket
67
68     try:
69         sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
70         sock.settimeout(2)
71         result = sock.connect_ex(('localhost', 5672))
72         sock.close()
73
74         assert result == 0, "RabbitMQ port 5672 is not accessible"
75     except Exception as e:
76         pytest.fail(f"Cannot reach RabbitMQ: {e}")
77
```

main* 0 0 0 Ln 5, Col 51 Spaces: 4 UTF-8 CRLF Python

File Edit Selection View Go Run Terminal Help
test_rabbitmq.py X
tests > test_rabbitmq.py
44 async def test_rabbitmq_multiple_messages(broker):
55 for msg in messages:
56 await broker.publish(msg, "multi_queue")
57
58 await asyncio.sleep(2)
59
60 assert len(received_messages) == len(messages)
61 assert received_messages == messages
62
63
64 def test_rabbitmq_service_running():
65 """Test if RabbitMQ service is accessible"""
66 import socket
67
68 try:
69 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
70 sock.settimeout(2)
71 result = sock.connect_ex(('localhost', 5672))
72 sock.close()
73
74 assert result == 0, "RabbitMQ port 5672 is not accessible"
75 except Exception as e:
76 pytest.fail(f"Cannot reach RabbitMQ: {e}")
77

main* 0 0 0 Ln 5, Col 51 Spaces: 4 UTF-8 CRLF Python

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(lab6) PS C:\Users\Anastasia\Desktop\application-development-homework\Lab6> pytest .\tests\test_rabbitmq.py -v
===== test session starts =====
platform win32 -- Python 3.13.7, pytest-9.0.2, pluggy-1.6.0 -- C:\Users\Anastasia\Desktop\application-development-homework\Lab6\.venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\Anastasia\Desktop\application-development-homework\Lab6
configfile: pyproject.toml
plugins: anyio-4.12.0, Faker-38.2.0, asyncio-1.3.0
asyncio: mode=Mode.AUTO, debug=False, asyncio_default_fixture_loop_scope=None, asyncio_default_test_loop_scope=function
collected 4 items

tests/test_rabbitmq.py::test_rabbitmq_connection PASSED [ 25%]
tests/test_rabbitmq.py::test_rabbitmq_publish_subscribe PASSED [ 50%]
tests/test_rabbitmq.py::test_rabbitmq_multiple_messages PASSED [ 75%]
tests/test_rabbitmq.py::test_rabbitmq_service_running PASSED [100%]

===== 4 passed in 4.59s =====
(lab6) PS C:\Users\Anastasia\Desktop\application-development-homework\Lab6>
```

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

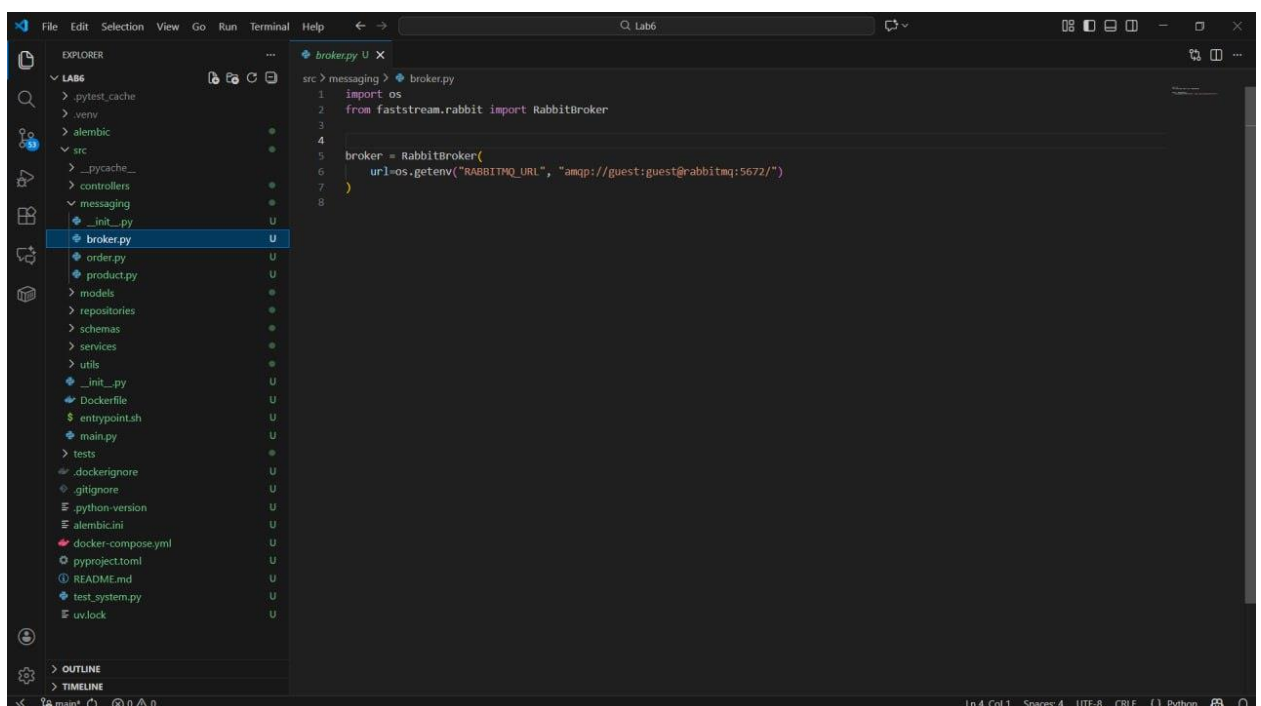
Используется фикстура `broker` для создания и очистки тестового брокера.

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:

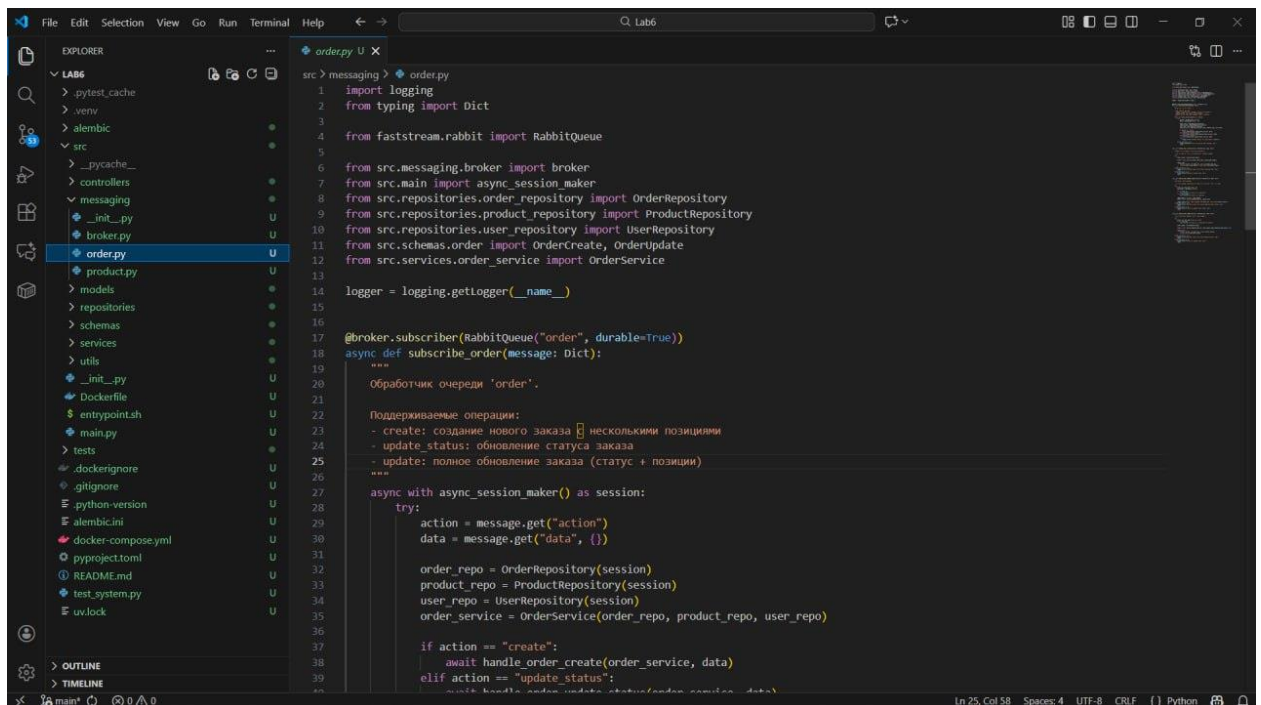
- Читает URL подключения из переменной окружения `RABBITMQ_URL`
- Используется для подписки на очереди и публикации сообщений

- 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`

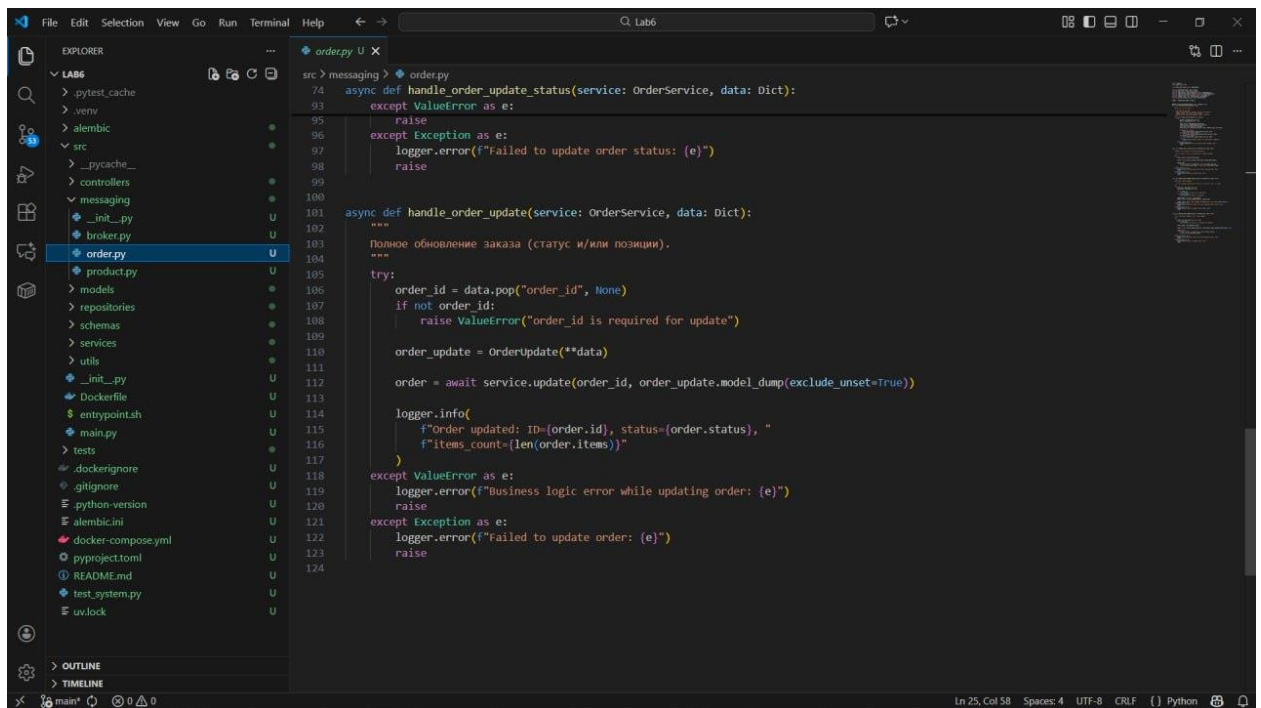
Файл: `src/messaging/order.py`



```
1 import logging
2 from typing import Dict
3
4 from faststream.rabbit import RabbitQueue
5
6 from src.messaging.broker import broker
7 from src.main import async_session_maker
8 from src.repositories.order_repository import OrderRepository
9 from src.repositories.product_repository import ProductRepository
10 from src.repositories.user_repository import UserRepository
11 from src.schemas.order import OrderCreate, OrderUpdate
12 from src.services.order_service import OrderService
13
14 logger = logging.getLogger(__name__)
15
16
17 @broker.subscriber(RabbitQueue("order", durable=True))
18 async def subscribe_order(message: Dict):
19     """
20     Обработчик очереди 'order'.
21
22     Поддерживаемые операции:
23     - create: создание нового заказа с несколькими позициями
24     - update status: обновление статуса заказа
25     - update: полное обновление заказа (статус + позиции)
26     """
27     async with async_session_maker() as session:
28         try:
29             action = message.get("action")
30             data = message.get("data", {})
31
32             order_repo = OrderRepository(session)
33             product_repo = ProductRepository(session)
34             user_repo = UserRepository(session)
35             order_service = OrderService(order_repo, product_repo, user_repo)
36
37             if action == "create":
38                 await handle_order_create(order_service, data)
39             elif action == "update_status":
40                 await handle_order_update_status(order_service, data)
```

```
src > messaging > order.py
18 async def subscribe_order(message: Dict):
27     async with async_session_maker() as session:
28         try:
36             if action == "create":
37                 await handle_order_create(order_service, data)
38             elif action == "update_status":
39                 await handle_order_update_status(order_service, data)
40             elif action == "update":
41                 await handle_order_update(order_service, data)
42             else:
43                 logger.error(f"Unknown action for order queue: {action}")
44
45         except Exception as e:
46             logger.exception(f"Error processing order message: {e}")
47             raise
48
49
50
51 async def handle_order_create(service: OrderService, data: Dict):
52     """
53     Создание нового заказа в нескольких позициях.
54
55     Проверяет наличие товаров на складе перед созданием заказа.
56     """
57     try:
58         order_create = OrderCreate(**data)
59
60         order = await service.create_order(order_create.model_dump())
61
62         logger.info(
63             f"Order created: ID={order.id}, user_id={order.user_id}, "
64             f"total={order.total_amount}, items_count={len(order.items)}")
65     except ValueError as e:
66         logger.error(f"Business logic error while creating order: {e}")
67         raise
68     except Exception as e:
69         logger.error(f"Failed to create order: {e}")
70         raise
71
```

```
src > messaging > order.py
51 async def handle_order_create(service: OrderService, data: Dict):
57     try:
66         except ValueError as e:
67             logger.error(f"Business logic error while creating order: {e}")
68             raise
69     except Exception as e:
70         logger.error(f"Failed to create order: {e}")
71         raise
72
73
74 async def handle_order_update_status(service: OrderService, data: Dict):
75     """
76     Обновление статуса заказа.
77
78     При отмене заказа (cancelled) автоматически возвращает товары на склад.
79     """
80     try:
81         order_id = data.get("order_id")
82         new_status = data.get("status")
83
84         if not order_id:
85             raise ValueError("order_id is required")
86         if not new_status:
87             raise ValueError("status is required")
88
89         update_data = {"status": new_status}
90         order = await service.update(order_id, update_data)
91
92         logger.info(f"Order status updated: ID={order.id}, new_status={order.status}")
93     except ValueError as e:
94         logger.error(f"Business logic error while updating order status: {e}")
95         raise
96     except Exception as e:
97         logger.error(f"Failed to update order status: {e}")
98         raise
99
100
101 async def handle_order_update(service: OrderService, data: Dict):
102     """
103     Полное обновление заказа (статус и/или позиции).
104
```



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

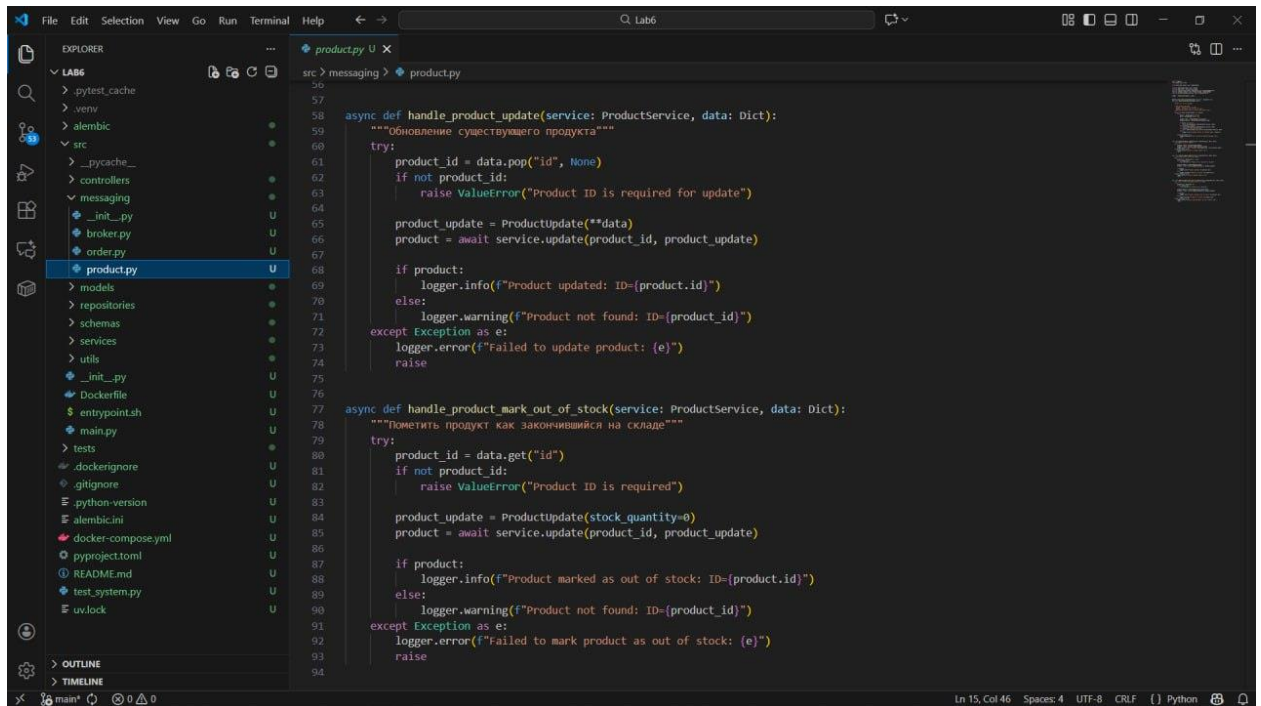
Файл: *src/messaging/product.py*

The image displays two screenshots of a Visual Studio Code editor window, showing the implementation of a product messaging service in Python. The editor is configured with a dark theme and the file explorer on the left shows the project structure, including files like `product.py`, `broker.py`, and `order.py`.

The first screenshot shows the initial setup of the `product.py` file. It includes imports for `logging`, `Dict`, and `RabbitQueue`. It defines a `logger` and a `@broker.subscriber` decorator for the `subscribe_product` function. The function handles messages from the 'product' queue and processes them based on the 'action' field: 'create', 'update', or 'mark_out_of_stock'. The `handle_product_create` function is also shown, which creates a new product using the `ProductCreate` schema and the `ProductService`.

The second screenshot shows the continuation of the implementation. It includes the `handle_product_update` function, which updates an existing product using the `ProductUpdate` schema and the `ProductService`. It also shows the `handle_product_mark_out_of_stock` function, which marks a product as out of stock. The code uses `async with async_session_maker() as session:` to manage database sessions. Error handling is implemented using `except Exception as e:` blocks, logging the error and raising it to be caught by the RabbitMQ consumer.

```
1 import logging
2 from typing import Dict
3
4 from faststream.rabbit import RabbitQueue
5
6 from src.messaging.broker import broker
7 from src.main import async_session_maker
8 from src.repositories.product_repository import ProductRepository
9 from src.schemas.product import ProductCreate, ProductUpdate
10 from src.services.product_service import ProductService
11
12 logger = logging.getLogger(__name__)
13
14 @broker.subscriber(queue=RabbitQueue("product", durable=True))
15 async def subscribe_product(message: Dict):
16     """
17     Обработчик очереди 'product'.
18
19     Поддерживаемые операции:
20     - create: создание нового продукта
21     - update: обновление существующего продукта
22     - mark_out_of_stock: пометить продукт как закончившийся
23     """
24     async with async_session_maker() as session:
25         try:
26             action = message.get("action")
27             data = message.get("data", {})
28
29             product_repo = ProductRepository(session)
30             product_service = ProductService(product_repo)
31
32             if action == "create":
33                 await handle_product_create(product_service, data)
34             elif action == "update":
35                 await handle_product_update(product_service, data)
36             elif action == "mark_out_of_stock":
37                 await handle_product_mark_out_of_stock(product_service, data)
38             else:
39                 logger.error(f"Unknown action for product queue: {action}")
40
41         except Exception as e:
42             logger.exception(f"Error processing product message: {e}")
43             raise
44
45 async def handle_product_create(service: ProductService, data: Dict):
46     """Создание нового продукта"""
47     try:
48         product_create = ProductCreate(**data)
49         product = await service.create(product_create)
50         logger.info(f"Product created: ID={product.id}, name={product.name}")
51     except Exception as e:
52         logger.error(f"Failed to create product: {e}")
53         raise
54
55 async def handle_product_update(service: ProductService, data: Dict):
56     """Обновление существующего продукта"""
57     try:
58         product_id = data.pop("id", None)
59         if not product_id:
60             raise ValueError("Product ID is required for update")
61
62         product_update = ProductUpdate(**data)
63         product = await service.update(product_id, product_update)
64
65         if product:
66             logger.info(f"Product updated: ID={product.id}")
67         else:
68             logger.warning(f"Product not found: ID={product_id}")
69     except Exception as e:
70         logger.error(f"Failed to update product: {e}")
71         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:

- Требуется 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)

```
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
```



```

141 @asynccontextmanager
142 async def lifespan(app: Litestar):
143     """
144     Управление жизненным циклом приложения.
145
146     Выполняется при старте и остановке приложения.
147     """
148     shutdown_event = asyncio.Event()
149     broker_task = asyncio.create_task(_broker_connect_loop(shutdown_event))
150
151     from src.messaging import order, product # noqa: F401
152
153     try:
154         yield
155     finally:
156         shutdown_event.set()
157
158         try:
159             await broker.close()
160         except Exception:
161             logger.exception("Error while closing broker")
162
163         broker_task.cancel()
164         try:
165             await broker_task
166         except asyncio.CancelledError:
167             pass
168         except Exception:
169             logger.exception("Broker task raised while shutting down")
170
171         try:
172             await engine.dispose()
173         except Exception:
174             logger.exception("Error disposing engine")
175

```

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:

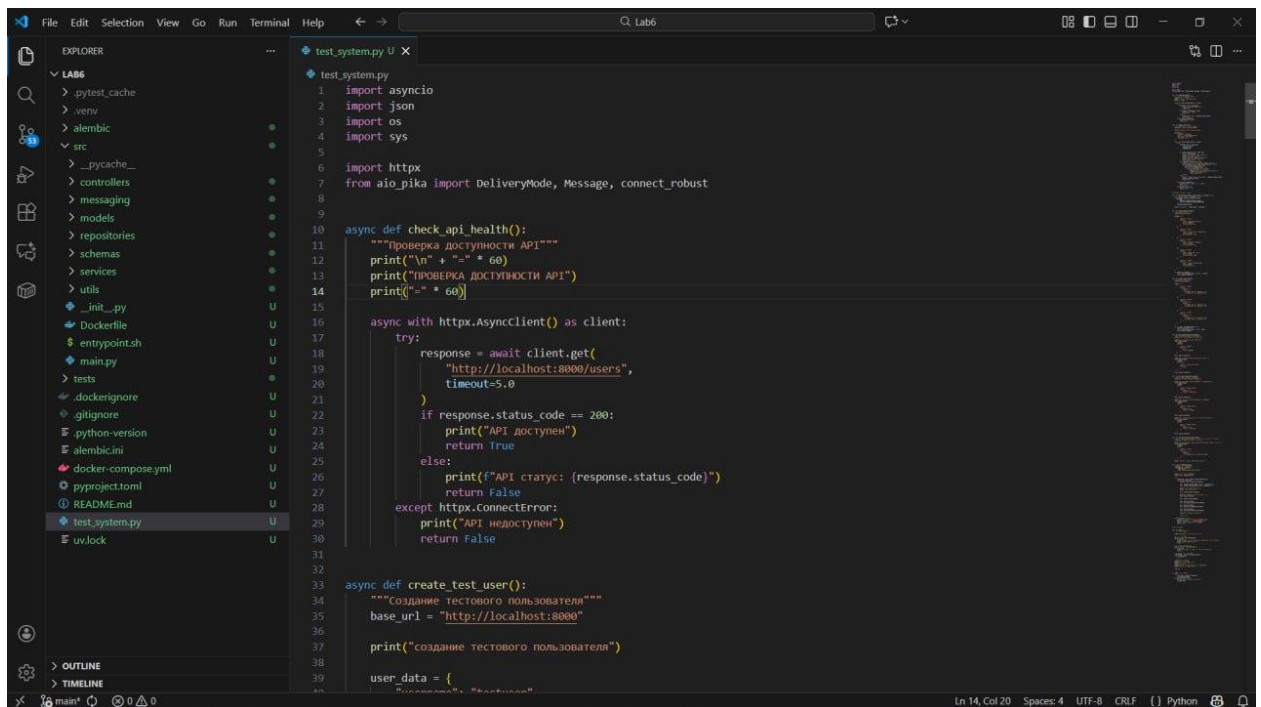
- Устанавливает флаг shutdown_event
- Отменяет фоновую задачу брокера
- Закрывает соединение с БД через engine.dispose()

Часть 4: Продюсер данных

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

Файл: *test_system.py*

(скрипт продюсера)



```
1 import asyncio
2 import json
3 import os
4 import sys
5
6 import httpx
7 from aio_pika import DeliveryMode, Message, connect_robust
8
9
10 async def check_api_health():
11     """Проверка доступности API"""
12     print("\n" + "-" * 60)
13     print("ПРОВЕРКА ДОСТУПНОСТИ API")
14     print("-" * 60)
15
16
17 async with httpx.AsyncClient() as client:
18     try:
19         response = await client.get(
20             "http://localhost:8000/users",
21             timeout=5.0
22         )
23         if response.status_code == 200:
24             print("API доступен")
25             return True
26         else:
27             print(f"API cratyc: {response.status_code}")
28             return False
29     except httpx.ConnectError:
30         print("API недоступен")
31         return False
32
33
34 async def create_test_user():
35     """Создание тестового пользователя"""
36     base_url = "http://localhost:8000"
37     print("создание тестового пользователя")
38
39     user_data = {
```

```
10 async def check_api_health():
31
32
33 async def create_test_user():
34     """Создание тестового пользователя"""
35     base_url = "http://localhost:8000"
36
37     print("создание тестового пользователя")
38
39     user_data = {
40         "username": "testuser",
41         "email": "testuser@example.com",
42         "full_name": "Test User",
43         "is_active": True
44     }
45
46     async with httpx.AsyncClient() as client:
47         try:
48             response = await client.post(
49                 f"{base_url}/users",
50                 json=user_data,
51                 timeout=10.0
52             )
53
54             if response.status_code in (200, 201):
55                 user = response.json()
56                 print(f"Пользователь создан успешно!")
57                 print(f"ID: {user['id']}")
58                 print(f"Username: {user['username']}")
59                 print(f"Email: {user['email']}")
60                 return user['id']
61             elif response.status_code == 409:
62                 print("Пользователь существует, получаем ID...")
63                 list_response = await client.get(f"{base_url}/users")
64                 if list_response.status_code == 200:
65                     users = list_response.json()
66                     for user in users.get('items', []):
67                         if user['username'] == 'testuser':
68                             print(f"Найден существующий пользователь!")
69                             print(f"ID: {user['id']}")
70                             return user['id']
71         except httpx.ConnectError:
72             print(f"Не удалось подключиться к API")
73             return None
74         except Exception as e:
75             print(f"Ошибка: {e}")
76             return None
77
78 # RabbitMQ: Отправка сообщений
79
80 async def send_message(channel, queue_name: str, message: dict):
81     """Отправка сообщения в указанную очередь"""
82     await channel.default_exchange.publish(
83         Message(
84             body=json.dumps(message).encode(),
85             delivery_mode=DeliveryMode.PERSISTENT,
```

```
85
86
87 # RabbitMQ: Отправка сообщений
88
89 async def send_message(channel, queue_name: str, message: dict):
90     """Отправка сообщения в указанную очередь"""
91     await channel.default_exchange.publish(
92         Message(
93             body=json.dumps(message).encode(),
94             delivery_mode=DeliveryMode.PERSISTENT,
```

The screenshot shows the VS Code editor with the file explorer on the left displaying a project structure. The main editor window shows the `test_system.py` file. The code includes an `async def send_message` function that publishes a message to a channel and an `async def create_products` function that creates a list of products. The status bar at the bottom indicates the cursor is at line 83, column 1.

```
83
84
85 async def send_message(channel, queue_name: str, message: dict):
86     """Отправка сообщения в указанную очередь"""
87     await channel.default_exchange.publish(
88         Message(
89             body=json.dumps(message).encode(),
90             delivery_mode=DeliveryMode.PERSISTENT,
91             routing_key=queue_name,
92         ),
93     )
94     print(f"Отправлено в '{queue_name}': {message}")
95
96
97 async def create_products(channel):
98     """Создание 5 продуктов"""
99     print("Создание продуктов")
100
101     products = [
102         {
103             "action": "create",
104             "data": {
105                 "name": "Laptop Dell XPS 15",
106                 "price": 1299.99,
107                 "stock_quantity": 25
108             }
109         },
110         {
111             "action": "create",
112             "data": {
113                 "name": "Wireless Mouse Logitech MX",
114                 "price": 79.99,
115                 "stock_quantity": 150
116             }
117         },
118         {
119             "action": "create",
120             "data": {
121                 "name": "Mechanical Keyboard",
122                 "price": 149.99,
```

This screenshot shows the same VS Code editor window, but the code in `test_system.py` has been updated. The `create_products` function now includes five product entries. The status bar at the bottom indicates the cursor is at line 83, column 1.

```
97 async def create_products(channel):
98     products = [
99         {
100             "data": {
101                 "price": 1299.99,
102                 "stock_quantity": 25
103             }
104         },
105         {
106             "action": "create",
107             "data": {
108                 "name": "Wireless Mouse Logitech MX",
109                 "price": 79.99,
110                 "stock_quantity": 150
111             }
112         },
113         {
114             "action": "create",
115             "data": {
116                 "name": "Mechanical Keyboard",
117                 "price": 149.99,
118                 "stock_quantity": 75
119             }
120         },
121         {
122             "action": "create",
123             "data": {
124                 "name": "USB-C Hub 7-in-1",
125                 "price": 49.99,
126                 "stock_quantity": 200
127             }
128         },
129         {
130             "action": "create",
131             "data": {
132                 "name": "Webcam Logitech C920",
133                 "price": 89.99,
134                 "stock_quantity": 5
135             }
136         }
137     ]
```

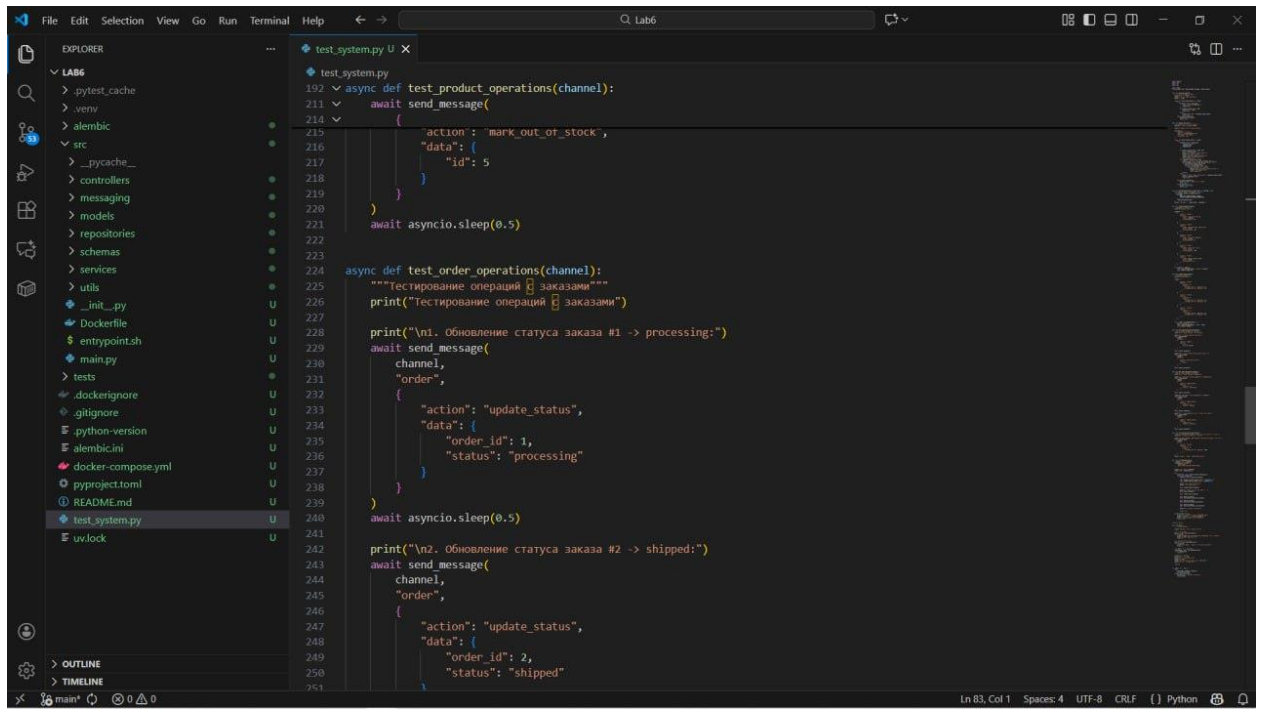
```
97 async def create_products(channel):
101     products = [
102         {
103             "data": {
104                 "price": 1299.99,
105                 "stock_quantity": 25
106             }
107         },
108         {
109             "action": "create",
110             "data": {
111                 "name": "Wireless Mouse Logitech MX",
112                 "price": 79.99,
113                 "stock_quantity": 150
114             }
115         },
116         {
117             "action": "create",
118             "data": {
119                 "name": "Mechanical Keyboard",
120                 "price": 149.99,
121                 "stock_quantity": 75
122             }
123         },
124         {
125             "action": "create",
126             "data": {
127                 "name": "USB-C Hub 7-in-1",
128                 "price": 49.99,
129                 "stock_quantity": 200
130             }
131         },
132         {
133             "action": "create",
134             "data": {
135                 "name": "Webcam Logitech C920",
136                 "price": 89.99,
137                 "stock_quantity": 5
138             }
139         }
140     ]
```

```
141     ]
142
143
144     for product in products:
145         await send_message(channel, "product", product)
146         await asyncio.sleep(0.5)
147
148
```



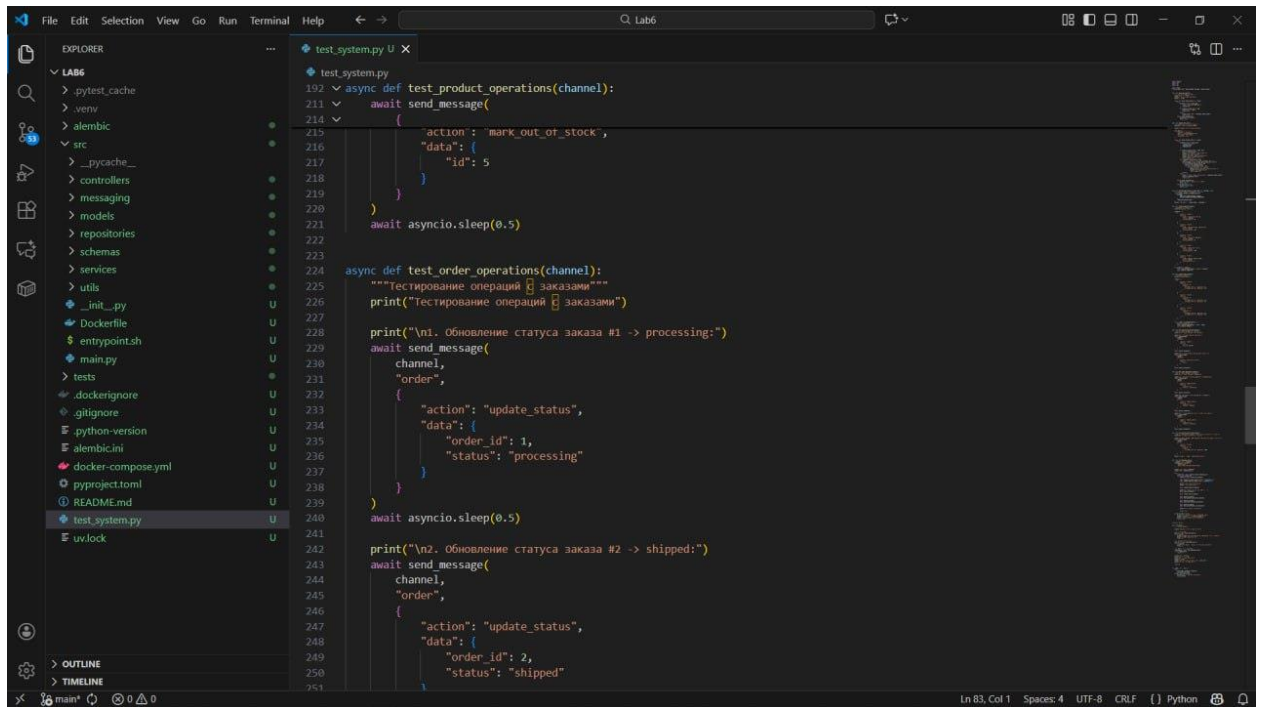
```
97 async def create_products(channel):
101     products = [
134         {
136             "data": {
140                 },
141             },
142     ]
143
144     for product in products:
145         await send_message(channel, "product", product)
146         await asyncio.sleep(0.5)
147
148
149 async def create_orders(channel):
150     """Создание 3 заказов"""
151     print("Создание заказов")
152
153     orders = [
154         {
155             "action": "create",
156             "data": {
157                 "user_id": 1,
158                 "items": [
159                     {"product_id": 1, "quantity": 1},
160                     {"product_id": 2, "quantity": 2},
161                 ]
162             },
163         },
164         {
165             "action": "create",
166             "data": {
167                 "user_id": 1,
168                 "items": [
169                     {"product_id": 3, "quantity": 1},
170                     {"product_id": 4, "quantity": 3},
171                 ]
172             },
173         },
174     ]
```

```
175
176
177     for i, order in enumerate(orders, 1):
178         print(f"№{i}")
179         await send_message(channel, "order", order)
180         await asyncio.sleep(1)
181
182
183
184
185
186
187
188
189
190
191
192 async def test_product_operations(channel):
193     """Тестирование операций с продуктами"""
194     print("Тестирование операций с продуктами")
195
```

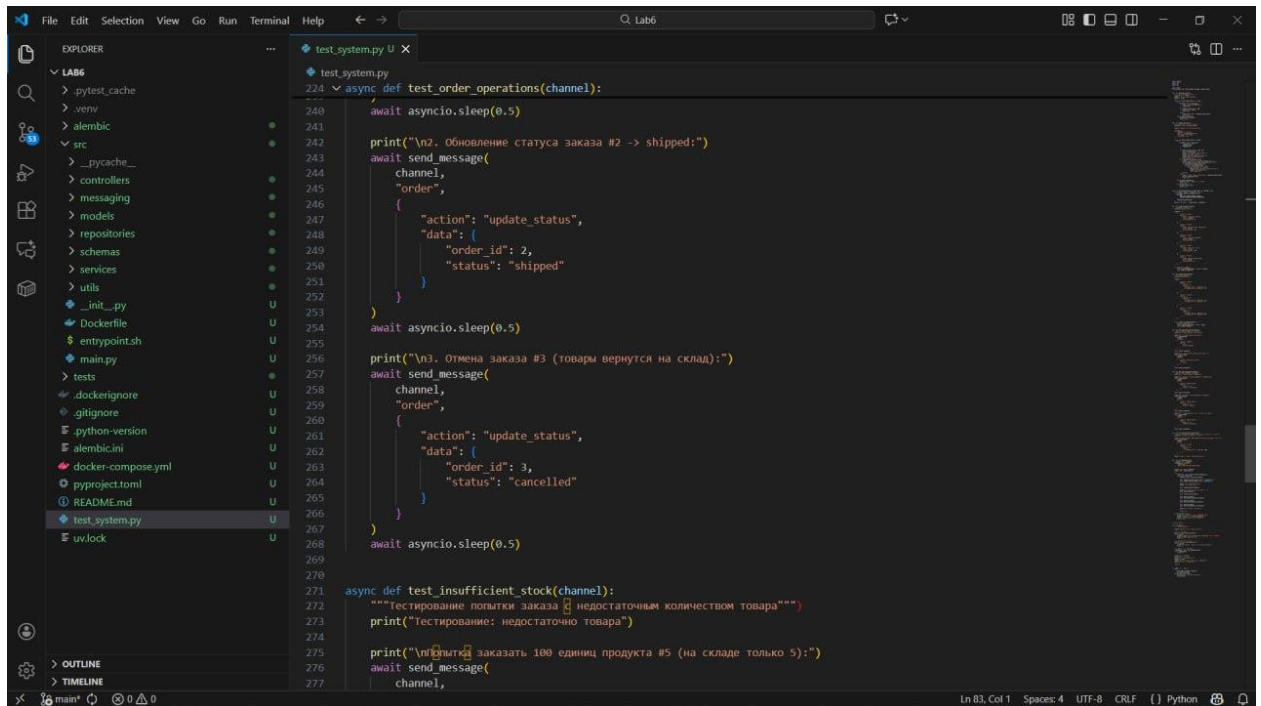
The screenshot shows the VS Code editor interface with the file explorer on the left and the code editor in the center. The file explorer shows a project structure with folders like .pytest_cache, .venv, alembic, src, and files like __init__.py, Dockerfile, entrypoint.sh, main.py, tests, .dockerignore, .gitignore, python-version, alembic.ini, docker-compose.yml, pyproject.toml, README.md, test_system.py, and uv.lock. The code editor displays the content of test_system.py, which includes two asynchronous functions: test_product_operations and test_order_operations. The test_order_operations function has a docstring in Russian and contains two print statements and two calls to send_message. The status of the order is updated from 'processing' to 'shipped'.

```
192 async def test_product_operations(channel):
193     await send_message(
194         {
195             "action": "mark_out_of_stock",
196             "data": {
197                 "id": 5
198             }
199         }
200     )
201     await asyncio.sleep(0.5)
202
203 async def test_order_operations(channel):
204     """Тестирование операций с заказами"""
205     print("Тестирование операций с заказами")
206
207     print("\n1. Обновление статуса заказа #1 -> processing:")
208     await send_message(
209         channel,
210         "order",
211         {
212             "action": "update_status",
213             "data": {
214                 "order_id": 1,
215                 "status": "processing"
216             }
217         }
218     )
219     await asyncio.sleep(0.5)
220
221     print("\n2. Обновление статуса заказа #2 -> shipped:")
222     await send_message(
223         channel,
224         "order",
225         {
226             "action": "update_status",
227             "data": {
228                 "order_id": 2,
229                 "status": "shipped"
230             }
231         }
232     )
```



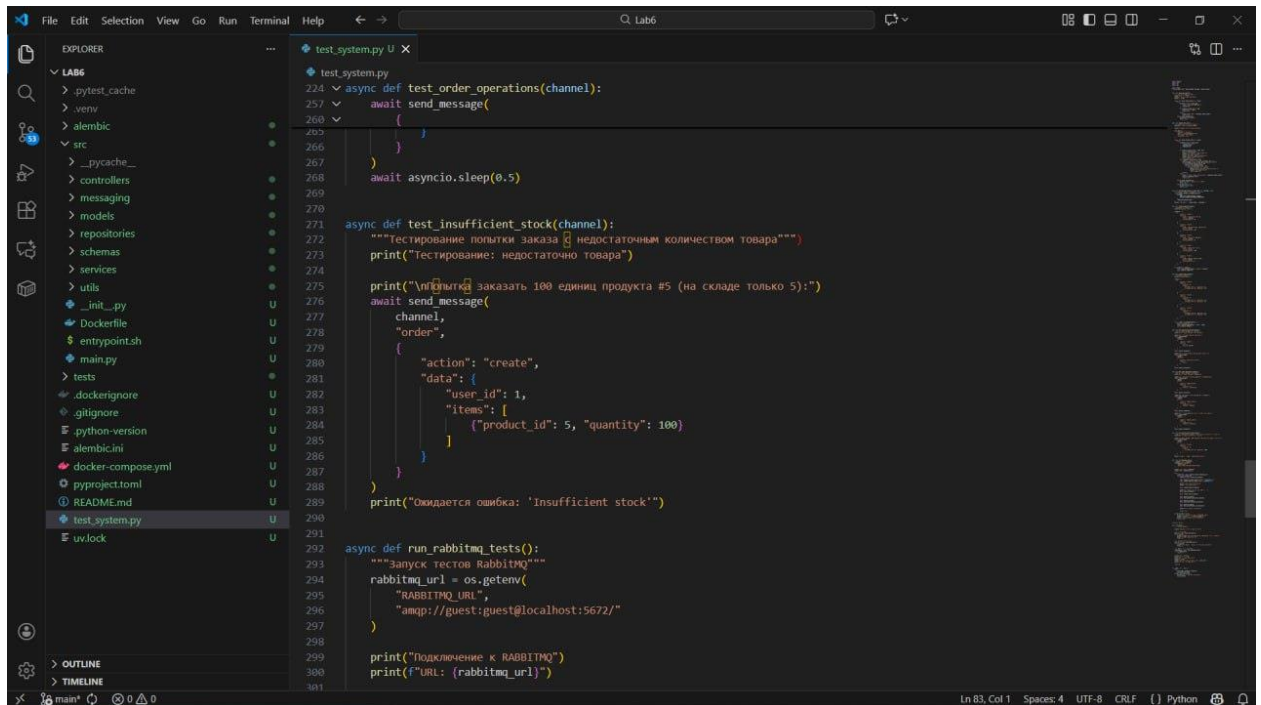
This screenshot is identical to the one above, showing the same VS Code editor interface and code content for test_system.py. The code defines two asynchronous functions: test_product_operations and test_order_operations. The test_order_operations function includes a Russian docstring, a print statement, and two calls to send_message to update the status of two orders from 'processing' to 'shipped'.

```
192 async def test_product_operations(channel):
193     await send_message(
194         {
195             "action": "mark_out_of_stock",
196             "data": {
197                 "id": 5
198             }
199         }
200     )
201     await asyncio.sleep(0.5)
202
203 async def test_order_operations(channel):
204     """Тестирование операций с заказами"""
205     print("Тестирование операций с заказами")
206
207     print("\n1. Обновление статуса заказа #1 -> processing:")
208     await send_message(
209         channel,
210         "order",
211         {
212             "action": "update_status",
213             "data": {
214                 "order_id": 1,
215                 "status": "processing"
216             }
217         }
218     )
219     await asyncio.sleep(0.5)
220
221     print("\n2. Обновление статуса заказа #2 -> shipped:")
222     await send_message(
223         channel,
224         "order",
225         {
226             "action": "update_status",
227             "data": {
228                 "order_id": 2,
229                 "status": "shipped"
230             }
231         }
232     )
```



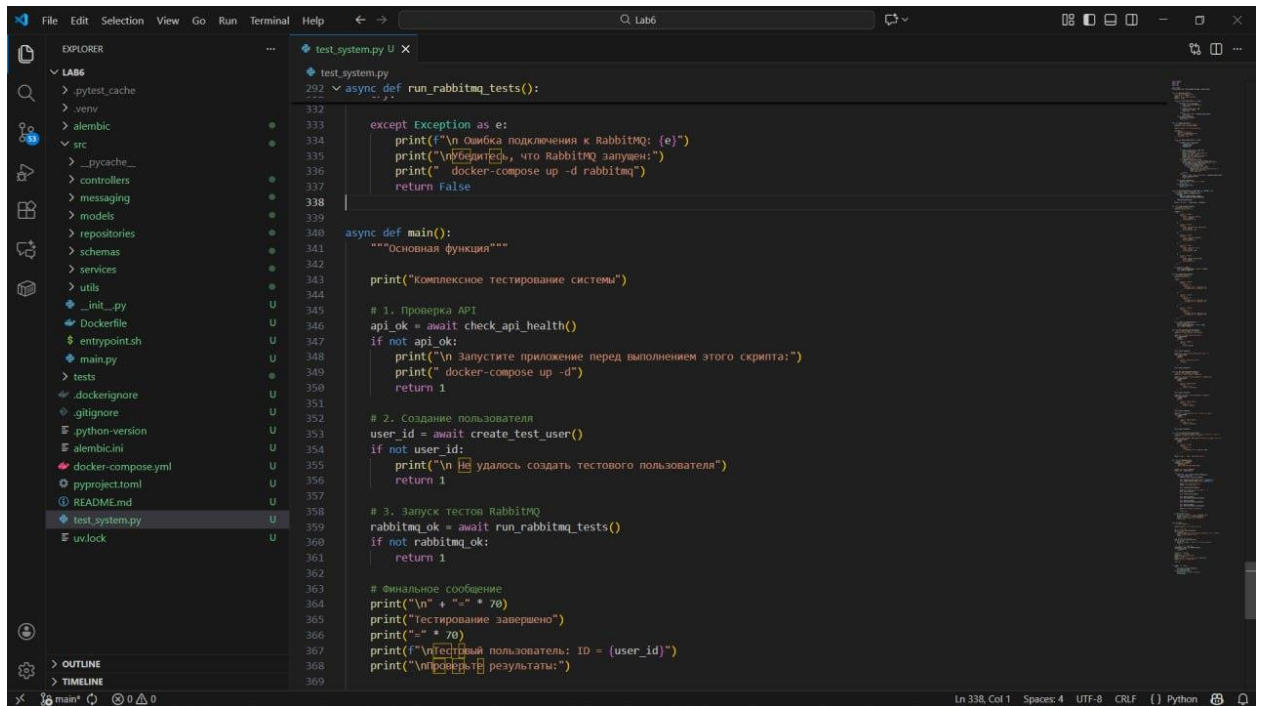
The screenshot shows the VS Code editor with the file explorer on the left displaying a project structure for 'LAB6'. The main editor window shows the file 'test_system.py' with the following code:

```
224 async def test_order_operations(channel):
240     await asyncio.sleep(0.5)
241
242     print("\n2. Обновление статуса заказа #2 -> shipped:")
243     await send_message(
244         channel,
245         "order",
246         {
247             "action": "update_status",
248             "data": {
249                 "order_id": 2,
250                 "status": "shipped"
251             }
252         }
253     )
254     await asyncio.sleep(0.5)
255
256     print("\n3. Отмена заказа #3 (товары вернутся на склад):")
257     await send_message(
258         channel,
259         "order",
260         {
261             "action": "update_status",
262             "data": {
263                 "order_id": 3,
264                 "status": "Cancelled"
265             }
266         }
267     )
268     await asyncio.sleep(0.5)
269
270
271 async def test_insufficient_stock(channel):
272     """Тестирование попытки заказа с недостаточным количеством товара"""
273     print("Тестирование: недостаточно товара")
274
275     print("\nПопытка заказать 100 единиц продукта #5 (на складе только 5):")
276     await send_message(
277         channel,
```



The screenshot shows the VS Code editor with the file explorer on the left displaying a project structure for 'LAB6'. The main editor window shows the file 'test_system.py' with the following code:

```
257     await send_message(
260         {
261         }
262     )
263     await asyncio.sleep(0.5)
264
265
266 async def test_insufficient_stock(channel):
267     """Тестирование попытки заказа с недостаточным количеством товара"""
268     print("Тестирование: недостаточно товара")
269
270     print("\nПопытка заказать 100 единиц продукта #5 (на складе только 5):")
271     await send_message(
272         channel,
273         "order",
274         {
275             "action": "create",
276             "data": {
277                 "user_id": 1,
278                 "items": [
279                     {"product_id": 5, "quantity": 100}
280                 ]
281             }
282         }
283     )
284     print("Ожидается ошибка: 'Insufficient stock'")
285
286
287 async def run_rabbitmq_tests():
288     """Запуск тестов RabbitMQ"""
289     rabbitmq_url = os.getenv(
290         "RABBITMQ_URL",
291         "amqp://guest:guest@localhost:5672/"
292     )
293
294     print("Подключение к RABBITMQ")
295     print(f"URL: {rabbitmq_url}")
296
297
298
299
300
301
```

```
292 async def run_rabbitmq_tests():
293     ...
332
333 except Exception as e:
334     print(f"\n Ошибка подключения к RabbitMQ: {e}")
335     print("\n Проверьте, что RabbitMQ запущен:")
336     print(" docker-compose up -d rabbitmq")
337     return False
338
339
340 async def main():
341     """Основная функция"""
342     print("Комплексное тестирование системы")
343
344     # 1. Проверка API
345     api_ok = await check_api_health()
346     if not api_ok:
347         print("\n Запустите приложение перед выполнением этого скрипта:")
348         print(" docker-compose up -d")
349         return 1
350
351     # 2. Создание пользователя
352     user_id = await create_test_user()
353     if not user_id:
354         print("\n Не удалось создать тестового пользователя")
355         return 1
356
357     # 3. Запуск тестов RabbitMQ
358     rabbitmq_ok = await run_rabbitmq_tests()
359     if not rabbitmq_ok:
360         return 1
361
362     # Финальное сообщение
363     print("\n" + "-" * 70)
364     print("Тестирование завершено")
365     print("-" * 70)
366     print(f"\n Тестовый пользователь: ID = {user_id}")
367     print("\n Проверьте результаты:")
368
369
```

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

```
(lab6) PS C:\Users\Anastasia\Desktop\application-development-homework\Lab6> python .\test_system.py
Комплексное тестирование системы
```

```
=====
ПРОВЕРКА ДОСТУПНОСТИ API
=====
```

```
API доступен
```

```
создание тестового пользователя
```

```
Пользователь существует, получаем ID...
```

```
Найден существующий пользователь!
```

```
ID: 1
```

```
Подключение к RABBITMQ
```

```
URL: amqp://guest:guest@localhost:5672/
```

```
Подключение установлено
```

```
Очереди объявлены
```

```
Создание продуктов
```

```
Отправлено в 'product': {'action': 'create', 'data': {'name': 'Laptop Dell XPS 15', 'price': 1299.99, 'stock_quantity': 25}}
```

```
Отправлено в 'product': {'action': 'create', 'data': {'name': 'Wireless Mouse Logitech MX', 'price': 79.99, 'stock_quantity': 150}}
```

```
Отправлено в 'product': {'action': 'create', 'data': {'name': 'Mechanical Keyboard', 'price': 149.99, 'stock_quantity': 75}}
```

```
Отправлено в 'product': {'action': 'create', 'data': {'name': 'USB-C Hub 7-in-1', 'price': 49.99, 'stock_quantity': 200}}
```

```
Отправлено в 'product': {'action': 'create', 'data': {'name': 'Webcam Logitech C920', 'price': 89.99, 'stock_quantity': 5}}
```

```
Ожидание обработки продуктов...
```

```
Создание заказов
```

```
Заказ #1:
```

```
Отправлено в 'order': {'action': 'create', 'data': {'user_id': 1, 'items': [{'product_id': 1, 'quantity': 1}, {'product_id': 2, 'quantity': 2}]}}
```

```
Заказ #2:
```

```
Отправлено в 'order': {'action': 'create', 'data': {'user_id': 1, 'items': [{'product_id': 3, 'quantity': 1}, {'product_id': 4, 'quantity': 3}]}}
```

```
Заказ #3:
```

```
Отправлено в 'order': {'action': 'create', 'data': {'user_id': 1, 'items': [{'product_id': 2, 'quantity': 1}, {'product_id': 5, 'quantity': 2}]}}
```

```
Тестирование операций с продуктами
```

```
1. Обновление цены продукта #1:
```

```
Отправлено в 'product': {'action': 'update', 'data': {'id': 1, 'price': 1199.99}}
```

```
2. Пометка продукта #5 как закончившегося:
```

```
Отправлено в 'product': {'action': 'mark_out_of_stock', 'data': {'id': 5}}
```

```
Тестирование операций с заказами
```

```
1. Обновление статуса заказа #1 -> processing:
```

```
Отправлено в 'order': {'action': 'update_status', 'data': {'order_id': 1, 'status': 'processing'}}
```

```
2. Обновление статуса заказа #2 -> shipped:
```

```
Отправлено в 'order': {'action': 'update_status', 'data': {'order_id': 2, 'status': 'shipped'}}
```

```
3. Отмена заказа #3 (товары вернутся на склад):
```

```
Отправлено в 'order': {'action': 'update_status', 'data': {'order_id': 3, 'status': 'cancelled'}}
```

```
Тестирование: недостаточно товара
```



```
3. Отмена заказа #3 (товары вернутся на склад):  
Отправлено в 'order': {'action': 'update_status', 'data': {'order_id': 3, 'status': 'cancelled'}}  
Тестирование: недостаточно товара  
  
Попытка заказать 100 единиц продукта #5 (на складе только 5):  
Отправлено в 'order': {'action': 'create', 'data': {'user_id': 1, 'items': [{'product_id': 5, 'quantity': 100}]}}  
Ожидается ошибка: 'Insufficient stock'  
Все сообщения отправлены  
  
=====
```

Тестирование завершено

=====

Тестовый пользователь: ID = 1

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

```

WHERE orders.id = $1::INTEGER
2025-12-07 20:11:37,470 INFO sqlalchemy.engine.Engine [cached since 1.109s ago] (3,)
INFO - 2025-12-07 20:11:37,469 - sqlalchemy.engine.Engine - base - SELECT orders.id
FROM orders
WHERE orders.id = $1::INTEGER
INFO - 2025-12-07 20:11:37,470 - sqlalchemy.engine.Engine - base - [cached since 1.109s ago] (3,)
2025-12-07 20:11:37,473 INFO sqlalchemy.engine.Engine SELECT order_items.id AS order_items_id, order_items.order_id AS order_items_order_id, or
der_items.product_id AS order_items_product_id, order_items.quantity AS order_items_quantity, order_items.unit_price AS order_items_unit_price
FROM order_items
WHERE $1::INTEGER = order_items.order_id
2025-12-07 20:11:37,473 INFO sqlalchemy.engine.Engine [cached since 1.105s ago] (3,)
INFO - 2025-12-07 20:11:37,473 - sqlalchemy.engine.Engine - base - SELECT order_items.id AS order_items_id, order_items.order_id AS order_items
_order_id, order_items.product_id AS order_items_product_id, order_items.quantity AS order_items_quantity, order_items.unit_price AS order_item
s_unit_price
FROM order_items
WHERE $1::INTEGER = order_items.order_id
INFO - 2025-12-07 20:11:37,473 - sqlalchemy.engine.Engine - base - [cached since 1.105s ago] (3,)
INFO - 2025-12-07 20:11:37,477 - src.messaging.order - order - Order status updated: ID=3, new_status=cancelled
2025-12-07 20:11:37,478 INFO sqlalchemy.engine.Engine ROLLBACK
INFO - 2025-12-07 20:11:37,478 - sqlalchemy.engine.Engine - base - ROLLBACK
2025-12-07 20:11:37,480 INFO - | order | 19c6d9a0bf - Processed
2025-12-07 20:11:39,981 INFO - | order | cd11f83c95 - Received
Looking for user with ID: 1 (type: <class 'int'>)

```

```

2025-12-07 20:11:39,984 INFO sqlalchemy.engine.Engine [cached since 11.92s ago] (1,)
WHERE users.id = $1::INTEGER
INFO - 2025-12-07 20:11:39,984 - sqlalchemy.engine.Engine - base - [cached since 11.92s ago] (1,)
Found user: User(id=1, username=testuser, email=testuser@example.com)
2025-12-07 20:11:39,987 INFO sqlalchemy.engine.Engine SELECT products.id AS products_id, products.name AS products_name, products.price AS prod
ucts_price, products.stock_quantity AS products_stock_quantity
FROM products
WHERE products.id = $1::INTEGER
2025-12-07 20:11:39,988 INFO sqlalchemy.engine.Engine [cached since 11.89s ago] (5,)
INFO - 2025-12-07 20:11:39,987 - sqlalchemy.engine.Engine - base - SELECT products.id AS products_id, products.name AS products_name, products.
price AS products_price, products.stock_quantity AS products_stock_quantity
FROM products
WHERE products.id = $1::INTEGER
INFO - 2025-12-07 20:11:39,988 - sqlalchemy.engine.Engine - base - [cached since 11.89s ago] (5,)
ERROR - 2025-12-07 20:11:39,990 - src.messaging.order - order - Business logic error while creating order: Insufficient stock
2025-12-07 20:11:40,012 INFO sqlalchemy.engine.Engine ROLLBACK
ERROR - 2025-12-07 20:11:39,990 - src.messaging.order - order - Error processing order message: Insufficient stock
Traceback (most recent call last):
  File "/app/src/messaging/order.py", line 44, in subscribe_order
    await handle_order_create(order_service, data)
  File "/app/src/messaging/order.py", line 66, in handle_order_create
    order = await service.create_order(order_create.model_dump())
    ~~~~~
  File "/app/src/services/order_service.py", line 46, in create_order
    raise ValueError("Insufficient stock")
ValueError: Insufficient stock
INFO - 2025-12-07 20:11:40,012 - sqlalchemy.engine.Engine - base - ROLLBACK
2025-12-07 20:11:40,014 ERROR - | order | cd11f83c95 - ValueError: Insufficient stock
Traceback (most recent call last):
  File "/app/.venv/lib/python3.13/site-packages/faststream/_internal/endpoint/subscriber/usecase.py", line 364, in process_message
    await h.call(

```

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,
      "status": "processing"
    }
  ]
}
```


Response body

```
    "unit_price": 79.99
  },
],
},
{
  "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,
  "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
    }
  ]
}
```

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:

- Стандартизированный 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.