

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

Институт радиоэлектроники и информационных технологий – РТФ

**Лабораторная работа №8:  
Основы работы с планировщиками**

Студент группы РИМ – 150950: \_\_\_\_\_ Вальнева А.Д.

Екатеринбург 2025

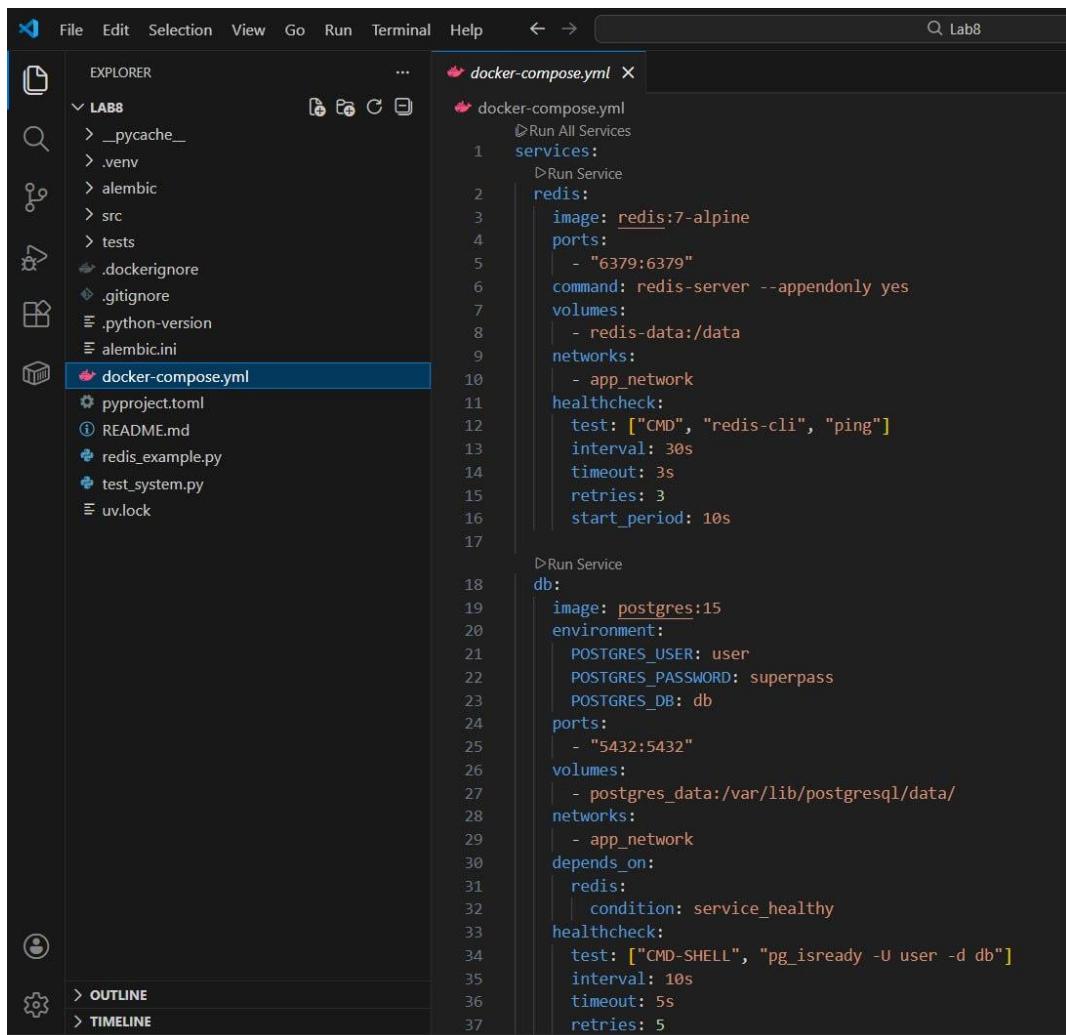
## Цель работы

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

## Задачи

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

## Часть 0: Настройка конфигурации Docker



```
File Edit Selection View Go Run Terminal Help < > Lab8

EXPLORER
LAB8
> __pycache__
> .venv
> alembic
> src
> tests
> .dockerignore
> .gitignore
> .python-version
> alembic.ini
docker-compose.yml
pyproject.toml
README.md
redis_example.py
test_system.py
uv.lock

dockerdocker-compose.yml
dockerdocker-compose.yml
Run All Services
services:
  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    command: redis-server --appendonly yes
    volumes:
      - redis-data:/data
    networks:
      - app_network
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 30s
      timeout: 3s
      retries: 3
      start_period: 10s
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: superpass
      POSTGRES_DB: db
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    networks:
      - app_network
    depends_on:
      redis:
        condition: service_healthy
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user -d db"]
      interval: 10s
      timeout: 5s
      retries: 5

OUTLINE
TIMELINE
```

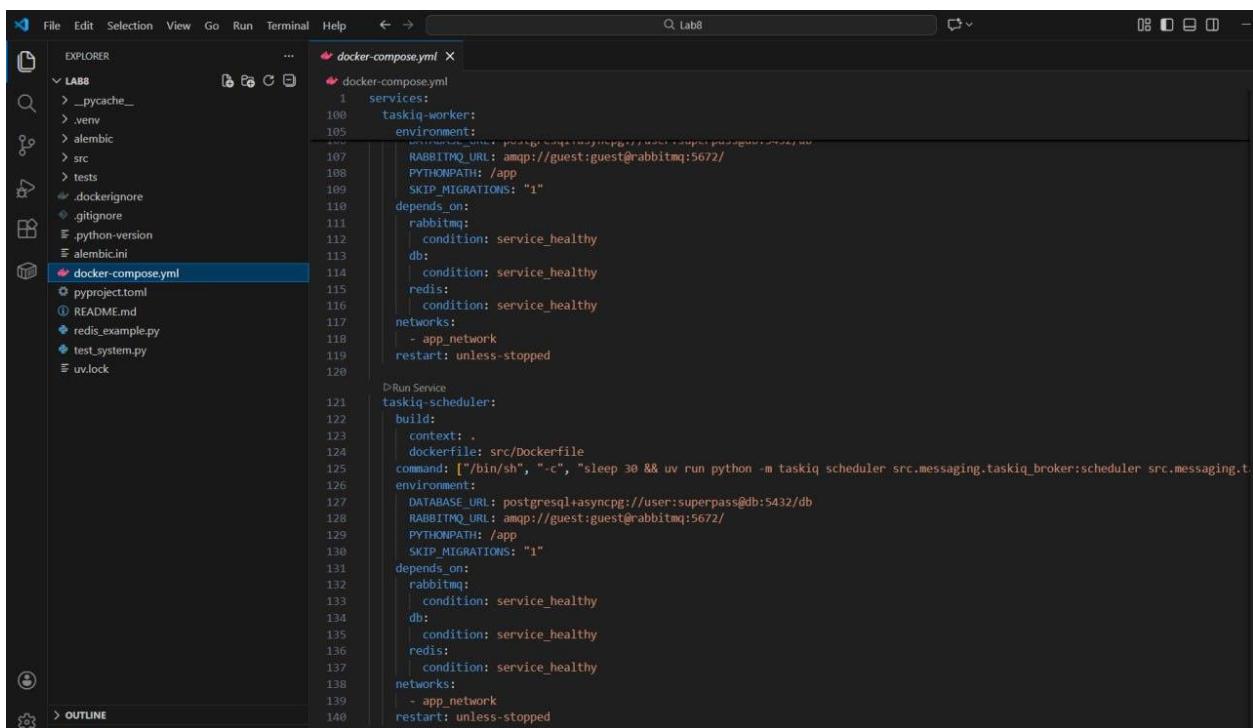
The screenshot shows the VS Code interface with the 'docker-compose.yml' file open in the right-hand editor pane. The file defines two services: 'redis' and 'db'. The 'redis' service uses the 'redis:7-alpine' image, exposes port 6379, runs a Redis server with append-only logging, and stores data in a volume named 'redis-data'. It also defines a network named 'app\_network' and includes a healthcheck for Redis. The 'db' service uses the 'postgres:15' image, sets environment variables for PostgreSQL user, password, and database, exposes port 5432, and maps a volume named 'postgres\_data' to the PostgreSQL data directory. It also includes a network named 'app\_network' and depends on the 'redis' service, with its own healthcheck.

The screenshot shows the VS Code interface with the docker-compose.yml file selected in the Explorer sidebar. The code editor displays the following configuration:

```
version: '3.8'
services:
  pgadmin:
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@example.com
      PGADMIN_DEFAULT_PASSWORD: superpass
    ports:
      - "8080:80"
    depends_on:
      db:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
      redis:
        condition: service_healthy
    networks:
      - app_network
  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"
    environment:
      - RABBITMQ_DEFAULT_VHOST=/
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq
    networks:
      - app_network
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: ["CMD", "rabbitmq-diagnostics", "check_port_connectivity"]
      interval: 5s
```

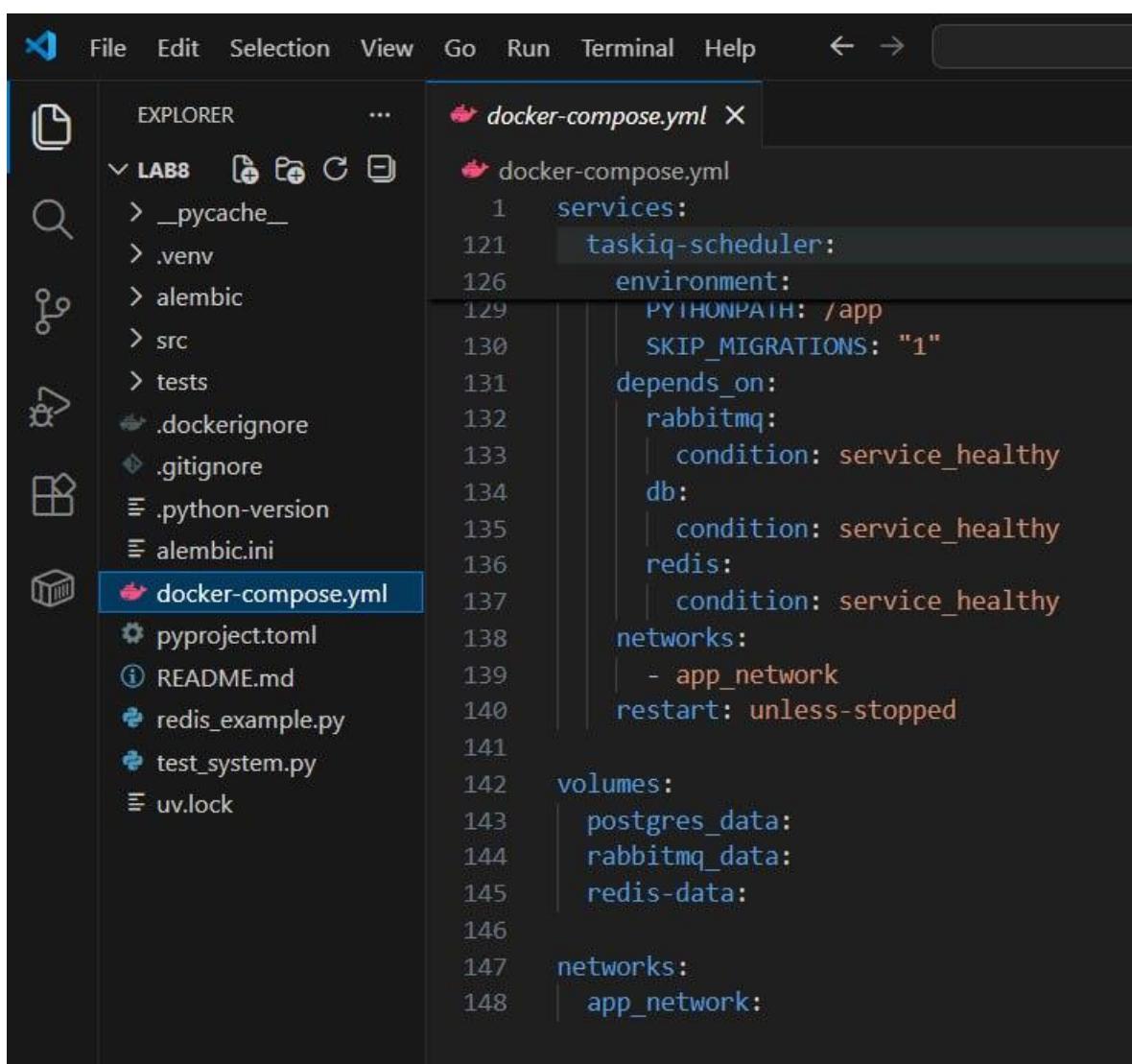
The screenshot shows the VS Code interface with the docker-compose.yml file selected in the Explorer sidebar. The code editor displays the following configuration, which includes additional services (app and taskiq-worker) and their configurations:

```
version: '3.8'
services:
  rabbitmq:
    healthcheck:
      timeout: 3s
      retries: 6
      start_period: 30s
  app:
    build:
      context: .
      dockerfile: src/Dockerfile
    environment:
      DATABASE_URL: postgresql+asyncpg://user:superpass@db:5432/db
      RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
      PYTHONPATH: /app
      START_TASKIQ_WORKER: 0
    ports:
      - "8000:8000"
    depends_on:
      db:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
      redis:
        condition: service_healthy
    networks:
      - app_network
  taskiq-worker:
    build:
      context: .
      dockerfile: src/Dockerfile
    command: ["bin/sh", "-c", "sleep 30 && uv run taskiq worker src.messaging.taskiq_broker:taskiq_broker src.messaging.tasks.repo"]
    environment:
      DATABASE_URL: postgresql+asyncpg://user:superpass@db:5432/db
      RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
      PYTHONPATH: /app
```



```
version: '3.8'
services:
  taskiq-worker:
    environment:
      RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
      PYTHONPATH: /app
      SKIP_MIGRATIONS: "1"
    depends_on:
      rabbitmq:
        condition: service_healthy
    db:
      condition: service_healthy
    redis:
      condition: service_healthy
    networks:
      - app_network
    restart: unless-stopped

  taskiq-scheduler:
    build:
      context: .
      dockerfile: src/Dockerfile
    command: ["bin/sh", "-c", "sleep 30 && uv run python -m taskiq scheduler src.messaging.taskiq_broker:scheduler src.messaging.t"]
    environment:
      DATABASE_URL: postgresql+asyncpg://user:superpass@db:5432/db
      RABBITMQ_URL: amqp://guest:guest@rabbitmq:5672/
      PYTHONPATH: /app
      SKIP_MIGRATIONS: "1"
    depends_on:
      rabbitmq:
        condition: service_healthy
      db:
        condition: service_healthy
    redis:
      condition: service_healthy
    networks:
      - app_network
    restart: unless-stopped
```



```
version: '3.8'
services:
  taskiq-scheduler:
    environment:
      PYTHONPATH: /app
      SKIP_MIGRATIONS: "1"
    depends_on:
      rabbitmq:
        condition: service_healthy
    db:
      condition: service_healthy
    redis:
      condition: service_healthy
    networks:
      - app_network
    restart: unless-stopped

volumes:
  postgres_data:
  rabbitmq_data:
  redis-data:

networks:
  app_network:
```

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

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

**Описание сервиса taskiq-worker:**

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

**Описание сервиса taskiq-scheduler:**

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

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

## **Часть 1: Создание миграций базы данных**

### **Ход выполнения**

Файл: *453659de82b0\_add\_timestamps\_to\_orders.py*

(Миграция добавления временных меток к заказам)



```
Lab8> alembic > versions > 453659de82b0_add_timestamps_to_orders.py U
 1  """add timestamps to orders
 2
 3  Revision ID: 453659de82b0
 4  Revises: 453659de82b0
 5  Create Date: 2025-12-12 17:32:58.553654
 6  """
 7  from typing import Sequence, Union
 8  from alembic import op
 9  import sqlalchemy as sa
10
11  # revision identifiers, used by Alembic.
12  revision: str = '453659de82b0'
13  down_revision: Union[str, Sequence[str], None] = 'a3853528df4f'
14  branch_labels: Union[str, Sequence[str], None] = None
15  depends_on: Union[str, Sequence[str], None] = None
16
17  def upgrade() -> None:
18      """Upgrade schema."""
19      op.add_column('orders', sa.Column('created_at', sa.DateTime(), server_default=sa.func.now(), nullable=False))
20      op.add_column('orders', sa.Column('updated_at', sa.DateTime(), server_default=sa.func.now(), nullable=False))
21
22  def downgrade() -> None:
23      """Downgrade schema."""
24      op.drop_column('orders', 'updated_at')
25      op.drop_column('orders', 'created_at')
```

**Создается view (представление) с нужными полями**

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

Цель: расширить таблицу orders полями created\_at и updated\_at для возможности корректной группировки и анализа заказов по датам, что является обязательным для формирования отчетов.

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

Файл: *4159f8afee9f\_create\_order\_reports\_view.py*  
(Миграция создания представления для отчетов)

```
4159f8afee9f_create_order_reports_view.py U X
Lab8 > alembic > versions > 4159f8afee9f_create_order_reports_view.py
1     """create order_reports view
2
3     Revision ID: 4159f8afee9f
4     Revises: a3853528df4f
5     Create Date: 2025-12-12 05:05:35.408287
6     """
7     from typing import Sequence, Union
8     from alembic import op
9     import sqlalchemy as sa
10
11    # revision identifiers, used by Alembic.
12    revision: str = '4159f8afee9f'
13    down_revision: Union[str, Sequence[str], None] = '453659de82b0'
14    branch_labels: Union[str, Sequence[str], None] = None
15    depends_on: Union[str, Sequence[str], None] = None
16
17
18    def upgrade() -> None:
19        """Upgrade schema."""
20        op.execute("""
21            CREATE VIEW order_reports AS
22            SELECT
23                o.created_at::DATE as report_at,
24                o.id as order_id,
25                COUNT(oi.product_id) as count_product,
26                o.total_amount as total_amount
27            FROM orders o
28            LEFT JOIN order_items oi ON oi.order_id = o.id
29            GROUP BY o.created_at::DATE, o.id, o.total_amount
30        """)
31
32
33    def downgrade() -> None:
34        """Downgrade schema."""
35        op.execute("DROP VIEW IF EXISTS order_reports")
```

## Создается view (представление) с нужными полями

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

### Описание:

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

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

## Часть 2: Создание модели OrderReport

Файл: *order\_report.py*

(Определена ORM-модель для работы с представлением SQLAlchemy)

The screenshot shows the PyCharm IDE interface. On the left is the 'EXPLORER' tool window displaying the project structure. It includes a tree view of files and folders under 'LAB8' and 'src'. Under 'src/models', there is a file named 'order\_report.py' which is currently selected and shown in the main code editor window. The code editor contains the following Python code:

```

from sqlalchemy import Column, Date, Integer, Float
from src.models.base import Base

class OrderReport(Base):
    """Представление для отчетов по заказам"""
    __tablename__ = "order_reports"
    __table_args__ = {'info': {'is_view': True}}

    report_at = Column(Date, primary_key=True, nullable=False)
    order_id = Column(Integer, primary_key=True, nullable=False)
    count_product = Column(Integer, nullable=False)
    total_amount = Column(Float, nullable=False)

    def __repr__(self) -> str:
        return (
            f"OrderReport(report_at={self.report_at}, "
            f"order_id={self.order_id}, count_product={self.count_product}, "
            f"total_amount={self.total_amount})"
        )

```

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

Порядок работы:

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

агgregирует данные из таблиц orders и order\_items.

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

## Часть 3: Создание репозитория для отчетов

Файл: *report repository.py*

```
src > repositories > report_repository.py
1  from datetime import date
2  from typing import List
3
4  from sqlalchemy import select
5  from sqlalchemy.ext.asyncio import AsyncSession
6
7  from src.models.order_report import OrderReport
8
9
10 class ReportRepository:
11     """Репозиторий для работы с отчетами."""
12
13     def __init__(self, session: AsyncSession):
14         self.session = session
15
16     async def get_report_by_date(self, report_date: date) -> List[OrderReport]:
17         """Получить отчет за конкретную дату."""
18         query = select(OrderReport).where(OrderReport.report_at == report_date)
19         result = await self.session.execute(query)
20         return list(result.scalars().all())
21
22     async def get_all_reports(self) -> List[OrderReport]:
23         """Получить все отчеты."""
24         query = select(OrderReport).order_by(
25             OrderReport.report_at.desc(),
26             OrderReport.order_id
27         )
28         result = await self.session.execute(query)
29         return list(result.scalars().all())
30
```

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

## Описание методов:

- `get_report_by_date()` - получает отчеты за конкретную дату с

фильтрацией по report\_at

- select(OrderReport) — использует модель представления.
- .where(OrderReport.report\_at == report\_date) - фильтрация по дате. Поле report\_at имеет тип DATE благодаря преобразованию DATE в SQL-представлении.
- get\_all\_reports() - получает все отчеты с сортировкой по дате (от новых к старым) и ID заказа
  - .order\_by(OrderReport.report\_at.desc(), OrderReport.order\_id) - сортировка сначала по дате отчета (новые сверху), затем по ID заказа для детерминированности.

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

#### **Часть 4: Создание сервиса для работы с отчетами**

Файл: *report\_service.py*

```

File Edit Selection View Go Run Terminal Help < > Q Lab8
EXPLORER report_service.py
src > services > report_service.py
1   from datetime import date
2   from typing import List, Dict, Any
3
4   from src.repositories.report_repository import ReportRepository
5
6
7 class ReportService:
8     """Сервис для работы с отчетами."""
9
10    def __init__(self, report_repository: ReportRepository):
11        self.report_repository = report_repository
12
13    def _serialize_report(self, report) -> Dict[str, Any]:
14        """Преобразует объект OrderReport в словарь."""
15        return {
16            "report_at": report.report_at.isoformat() if report.report_at else None,
17            "order_id": report.order_id,
18            "count_product": report.count_product,
19            "total_amount": float(report.total_amount) if report.total_amount else 0.0,
20        }
21
22    async def get_report_by_date(self, report_date: date) -> List[Dict[str, Any]]:
23        """
24        Получить отчет за конкретную дату.
25
26        Args:
27            report_date: Дата отчета
28
29        Returns:
30            Список словарей с данными отчетов
31        """
32        reports = await self.report_repository.get_report_by_date(report_date)
33        return [self._serialize_report(report) for report in reports]
34
35    async def get_all_reports(self) -> List[Dict[str, Any]]:
36        """
37        Получить все отчеты.
38
39        Returns:
40            Список словарей с данными отчетов
41        """
42        reports = await self.report_repository.get_all_reports()
43        return [self._serialize_report(report) for report in reports]
44

```

```

File Edit Selection View Go Run Terminal Help < > Q Lab8
EXPLORER report_service.py
src > services > report_service.py
7 class ReportService:
22     async def get_report_by_date(self, report_date: date) -> List[Dict[str, Any]]:
28
29         Returns:
30             Список словарей с данными отчетов
31         """
32         reports = await self.report_repository.get_report_by_date(report_date)
33         return [self._serialize_report(report) for report in reports]
34
35     async def get_all_reports(self) -> List[Dict[str, Any]]:
36
37         Получить все отчеты.
38
39         Returns:
40             Список словарей с данными отчетов
41         """
42         reports = await self.report_repository.get_all_reports()
43         return [self._serialize_report(report) for report in reports]
44

```

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

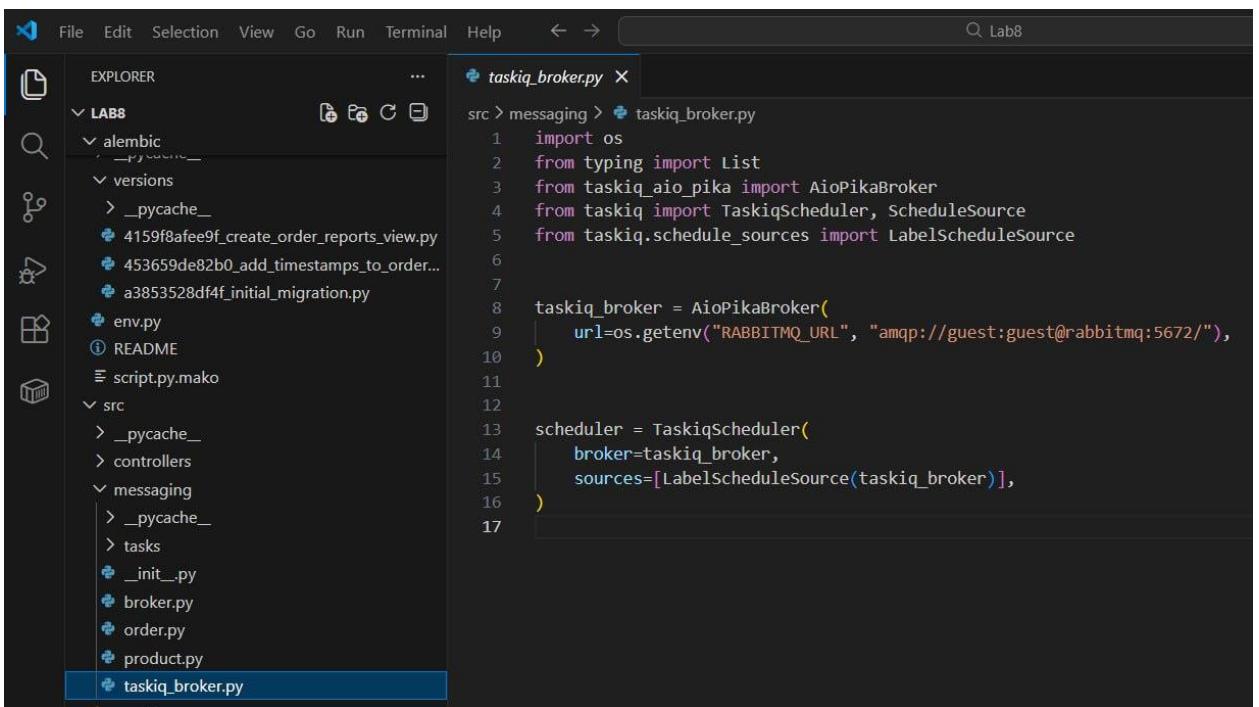
контроллером и репозиторием.

Описание работы методов:

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

## Часть 5: Создание планировщика

### Файл: *taskiq\_broker.py*



```
src > messaging > taskiq_broker.py
1 import os
2 from typing import List
3 from taskiq_aio_pika import AioPikaBroker
4 from taskiq import TaskiqScheduler, ScheduleSource
5 from taskiq.schedule_sources import LabelscheduleSource
6
7
8 taskiq_broker = AioPikaBroker(
9     url=os.getenv("RABBITMQ_URL", "amqp://guest:guest@rabbitmq:5672/"),
10 )
11
12 scheduler = TaskiqScheduler(
13     broker=taskiq_broker,
14     sources=[LabelscheduleSource(taskiq_broker)],
15 )
16
17
```

### Ход выполнения:

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

архитектуры и повторного использования.

Описание работы:

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

Запуск компонентов (через Docker Compose):

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

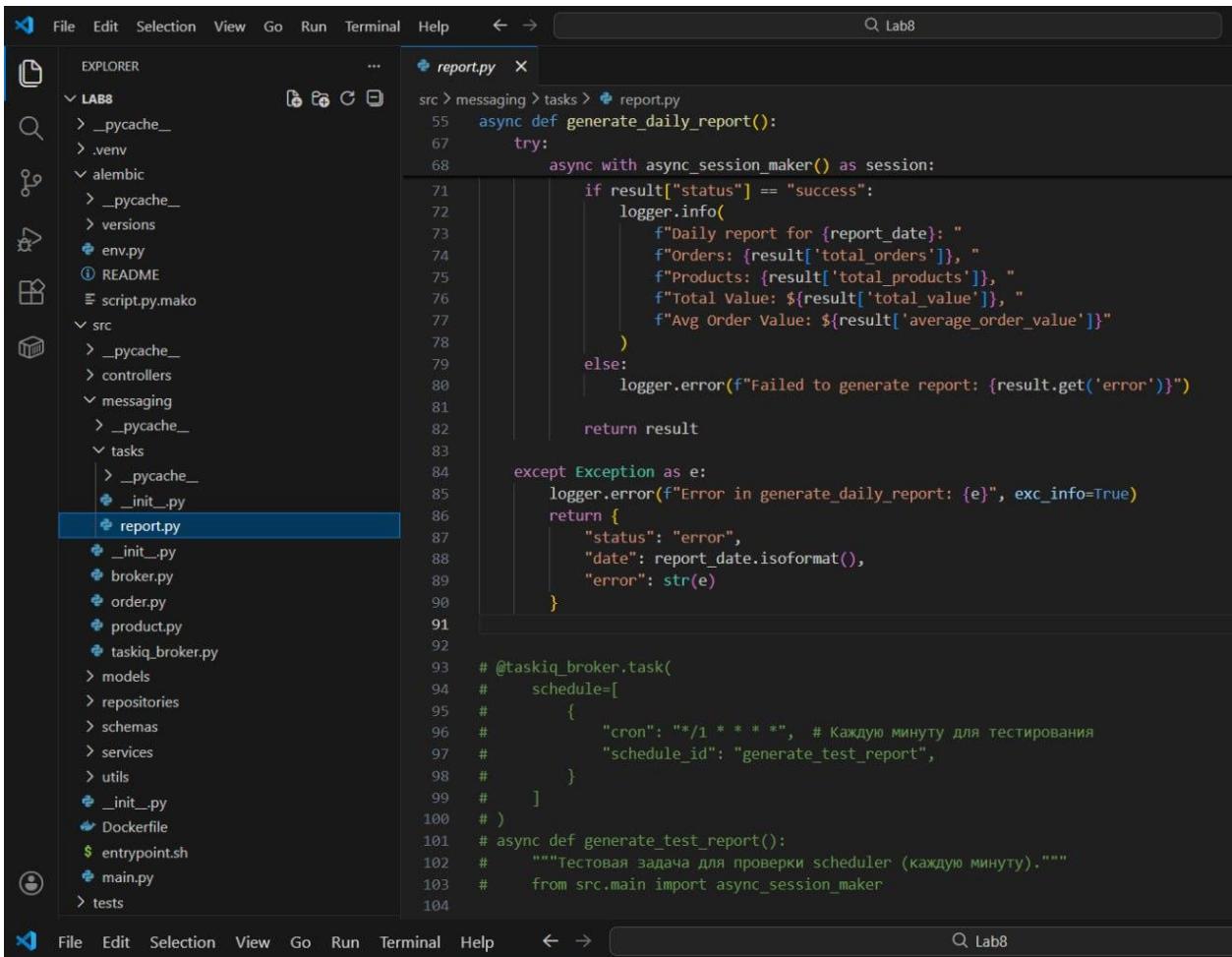
Файл: *report.py*  
(Реализация задачи генерации отчетов)

The screenshot shows a code editor interface with a dark theme. On the left is a sidebar with navigation icons (File, Edit, Selection, View, Go, Run, Terminal, Help) and a tree view of the project structure. The tree view shows a directory structure under 'LABB' with files like '\_pycache\_','.venv','alembic','versions','env.py','README','script.py.mako', and several files under 'src' including '\_pycache\_','controllers','messaging','tasks', and 'report.py'. The file 'report.py' is currently selected and open in the main editor area.

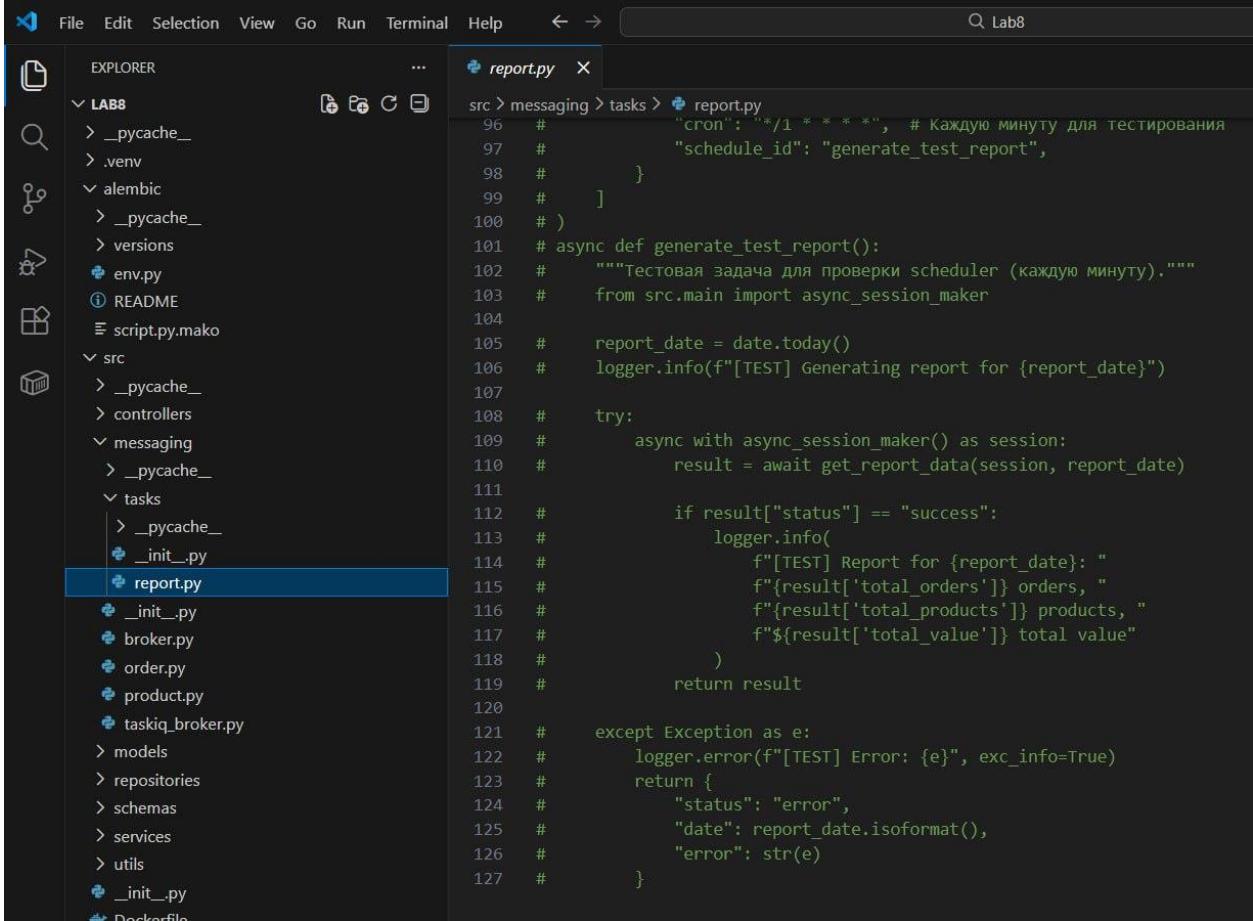
The main editor area displays the following Python code:

```
src > messaging > tasks > report.py
1 import logging
2 from datetime import date, datetime, timedelta
3
4 from sqlalchemy.ext.asyncio import AsyncSession
5
6 from src.messaging.taskiq_broker import taskiq_broker
7 from src.repositories.report_repository import ReportRepository
8 from src.services.report_service import ReportService
9
10 logger = logging.getLogger(__name__)
11
12
13 async def get_report_data(session: AsyncSession, report_date: date) -> dict:
14     """Получение данных отчета за указанную дату."""
15     try:
16         report_repository = ReportRepository(session)
17         report_service = ReportService(report_repository)
18
19         reports = await report_service.get_report_by_date(report_date)
20
21         total_orders = len(reports)
22         total_products = sum(report["count_product"] for report in reports)
23         total_value = sum(report["total_amount"] for report in reports)
24
25         avg_products = total_products / total_orders if total_orders > 0 else 0
26         avg_order_value = total_value / total_orders if total_orders > 0 else 0
27
28
29     return {
30         "status": "success",
31         "date": report_date.isoformat(),
32         "total_orders": total_orders,
33         "total_products": total_products,
34         "total_value": round(total_value, 2),
35         "average_products_per_order": round(avg_products, 2),
36         "average_order_value": round(avg_order_value, 2),
37         "reports": reports
38     }
```

```
src > messaging > tasks > report.py
13     async def get_report_data(session: AsyncSession, report_date: date) -> dict:
14         try:
15             ...
16         except Exception as e:
17             logger.error(f"Error getting report data: {e}", exc_info=True)
18             return {
19                 "status": "error",
20                 "error": str(e),
21                 "date": report_date.isoformat()
22             }
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47 @taskiq_broker.task(
48     schedule=[
49         {
50             "cron": "0 0 * * *", # Каждый день в полночь
51             "schedule_id": "generate_daily_report",
52         }
53     ]
54 )
55     async def generate_daily_report():
56         """
57             Формирует отчет по заказам за предыдущий день.
58             Выполняется автоматически каждый день в полночь.
59         """
60         from src.main import async_session_maker
61
62         # Отчет за вчерашний день
63         report_date = (datetime.now() - timedelta(days=1)).date()
64
65         logger.info(f"Starting daily report generation for {report_date}")
66
67         try:
68             async with async_session_maker() as session:
69                 result = await get_report_data(session, report_date)
70
71                 if result["status"] == "success":
72                     logger.info(
73                         f"Daily report for {report_date}: "
74                     )
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
```



```
src > messaging > tasks > report.py
55     async def generate_daily_report():
56         try:
57             async with async_session_maker() as session:
58                 if result["status"] == "success":
59                     logger.info(
60                         f"Daily report for {report_date}: "
61                         f"Orders: {result['total_orders']}, "
62                         f"Products: {result['total_products']}, "
63                         f"Total Value: ${result['total_value']}, "
64                         f"Avg Order Value: ${result['average_order_value']}"
65                     )
66                 else:
67                     logger.error(f"Failed to generate report: {result.get('error')}")
68
69             return result
70
71         except Exception as e:
72             logger.error(f"Error in generate_daily_report: {e}", exc_info=True)
73             return {
74                 "status": "error",
75                 "date": report_date.isoformat(),
76                 "error": str(e)
77             }
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93     # @taskiq_broker.task(
94     #     schedule=[
95     #         {
96     #             "cron": "*/* * * * *", # Каждую минуту для тестирования
97     #             "schedule_id": "generate_test_report",
98     #         }
99     #     ]
100    # )
101    # async def generate_test_report():
102    #     """Тестовая задача для проверки scheduler (каждую минуту)."""
103    #     from src.main import async_session_maker
104
```



```
src > messaging > tasks > report.py
96     #         "cron": "*/* * * * *", # Каждую минуту для тестирования
97     #         "schedule_id": "generate_test_report",
98     #     ]
99     #
100    #
101    # async def generate_test_report():
102    #     """Тестовая задача для проверки scheduler (каждую минуту)."""
103    #     from src.main import async_session_maker
104
105    #     report_date = date.today()
106    #     logger.info(f"[TEST] Generating report for {report_date}")
107
108    #     try:
109    #         async with async_session_maker() as session:
110    #             result = await get_report_data(session, report_date)
111
112    #             if result["status"] == "success":
113    #                 logger.info(
114    #                     f"[TEST] Report for {report_date}: "
115    #                     f"{result['total_orders']} orders, "
116    #                     f"{result['total_products']} products, "
117    #                     f"${result['total_value']} total value"
118    #                 )
119    #             return result
120
121    #     except Exception as e:
122    #         logger.error(f"[TEST] Error: {e}", exc_info=True)
123    #         return {
124    #             "status": "error",
125    #             "date": report_date.isoformat(),
126    #             "error": str(e)
127    #         }
```

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

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

**Задача `generate_daily_report()`** - точка входа, управляемая планировщиком.

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

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

## **Часть 6: Создание отчета по заказам**

## Файл: *report\_controller.py*

## (Создание REST API контроллера)

```
src > controllers > report_controller.py
1   from datetime import date
2   from typing import List, Dict, Any, Optional
3
4   from litestar import Controller, get
5   from litestar.params import Parameter
6   from litestar.datastructures import ResponseHeader
7   from litestar.exceptions import NotFoundException
8   from litestar.status_codes import HTTP_200_OK
9
10  from src.services.report_service import ReportService
11
12
13  class ReportController(Controller):
14      """Контроллер для работы с отчетами по заказам."""
15
16      path = "/report"
17      tags = ["reports"]
18
19  @get(
20      "/",
21      summary="Получить отчеты по заказам",
22      description=(
23          "Возращает отчет за конкретную дату или все отчеты, если дата не указана."
24          "Отчет формируется автоматически из таблиц orders и order_items."
25      ),
26      status_code=HTTP_200_OK,
27  )
28  async def get_reports(
29      self,
30      report_service: ReportService,
31      report_date: Optional[date] = Parameter(
32          default=None,
33          query="date",
34          description="Дата отчета в формате YYYY-MM-DD",
35          required=False,
36      ),
37  ) -> List[Dict[str, Any]]:
38      ...

```

```
File Edit Selection View Go Run Terminal Help < > Q Lab8  
EXPLORER ... report_controller.py X  
LABS > _pycache_ src > controllers > report_controller.py  
> .venv 13 class ReportController(Controller):  
> alembic 37 ) -> List[Dict[str, Any]]:  
> versions 38 """  
env.py 39 Получить отчеты по заказам.  
README 40  
script.py.mako 41  
src 42 Args:  
> _pycache_ 43 report_service: Сервис для работы с отчетами  
> controllers 44 report_date: Опциональная дата для фильтрации  
> _pycache_ 45 Returns:  
_init_.py 46 List[Dict[str, Any]]: Список отчетов с полями:  
order_controller.py 47 - report_at: дата отчета  
product_controller.py 48 - order_id: ID заказа  
report_controller.py 49 - count_product: количество продуктов в заказе  
user_controller.py 50 - total_amount: общая стоимость заказа  
messaging 51 Raises:  
tasks 52 NotFoundException: Если отчеты не найдены  
report.py 53 Examples:  
_init_.py 54 GET /report/ # Все отчеты  
broker.py 55 GET /report/?date=2024-12-12 # Отчеты за 12 декабря 2024  
order.py 56 """  
product.py 57 if report_date:  
taskiq_broker.py 58     reports = await report_service.get_report_by_date(report_date)  
models 59     if not reports:  
repositories 60         raise NotFoundException(  
schemas 61             detail=f"Отчеты за дату {report_date.isoformat()} не найдены"  
services 62     )  
OUTLINE 63 else:  
TIMELINE 64     reports = await report_service.get_all_reports()  
65     if not reports:  
66         raise NotFoundException(detail="Отчеты не найдены")  
67 return reports  
68 @get(  
69     "/summary",  
70     summary="Получить сводку по отчету",  
71     )  
72     
```

```
File Edit Selection View Go Run Terminal Help < > Q Lab8

EXPLORER
LABB
src > controllers > report_controller.py
13  class ReportController(Controller):
72    @get(
74        summary="Получить сводку по отчету",
75        description="Возвращает агрегированную статистику за указанную дату",
76        status_code=HTTP_200_OK,
77    )
78    async def get_report_summary(
79        self,
80        report_service: ReportService,
81        report_date: date = Parameter(
82            query="date",
83            description="Дата отчета в формате YYYY-MM-DD",
84            required=True,
85        ),
86    ) -> Dict[str, Any]:
87        """
88        Получить сводку по отчету за дату.
89
90        Args:
91            report_service: Сервис для работы с отчетами
92            report_date: Дата отчета
93
94        Returns:
95            Dict[str, Any]: Сводная статистика:
96                - date: дата отчета
97                - total_orders: всего заказов
98                - total_products: всего продуктов
99                - average_products_per_order: среднее кол-во продуктов на заказ
100               - total_value: общая стоимость всех заказов за дату
101               - average_order_value: средняя стоимость заказа
102
103        Raises:
104            NotFoundException: Если отчеты за дату не найдены
105
106        Example:
107            GET /report/summary?date=2024-12-12
108
109        reports = await report_service.get_report_by_date(report_date)
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
```

```
File Edit Selection View Go Run Terminal Help < > Q Lab8

EXPLORER
LABB
src > controllers > report_controller.py
13  class ReportController(Controller):
86    ) -> Dict[str, Any]:
102
103        Raises:
104            NotFoundException: Если отчеты за дату не найдены
105
106        Example:
107            GET /report/summary?date=2024-12-12
108
109        reports = await report_service.get_report_by_date(report_date)
110
111        if not reports:
112            raise NotFoundException(
113                detail=f"Отчеты за дату {report_date.isoformat()} не найдены"
114            )
115
116        total_orders = len(reports)
117        total_products = sum(r["count_product"] for r in reports)
118        total_value = sum(r["total_amount"] for r in reports)
119
120
121
122
123
124
125
126
```

## Ход выполнения:

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

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

### **Ключевые эндпоинты:**

#### **1) GET /report/**

- возвращает либо все отчеты, либо отчет за конкретную дату
- параметр date в запросе не обязательен, если он указан - контроллер вызывает report\_service.get\_report\_by\_date(), а если нет - report\_service.get\_all\_reports()
- инжекция зависимости report\_service через Litestar DI
- при отсутствии данных возвращается ошибка NotFoundException (статус 404), а не пустой список

#### **2) GET /report/summary**

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

Файл: *main.py*

**(Интеграция в главный модуль приложения)**

### **Ход выполнения:**

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

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer (Left):** Shows the project structure under the folder `LAB8`. The `src` folder contains subfolders `controllers`, `messaging`, `tasks`, and several files like `product_controller.py`, `report_controller.py`, etc.
- Editor (Right):** The file `main.py` is open. The code imports various modules from the `src` directory and defines a `DatabaseConfig` class.

```
src > main.py
1 import asyncio
2 import logging
3 import os
4 from contextlib import asynccontextmanager
5 from typing import AsyncGenerator
6
7 from litestar import Litestar
8 from litestar.di import Provide
9 from redis.asyncio import Redis
10 from sqlalchemy.ext.asyncio import (
11     AsyncEngine,
12     AsyncSession,
13     async_sessionmaker,
14     create_async_engine,
15 )
16
17 from src.controllers.order_controller import OrderController
18 from src.controllers.product_controller import ProductController
19 from src.controllers.user_controller import UserController
20 from src.controllers.report_controller import ReportController
21 from src.messaging.broker import broker as faststream_broker
22 from src.messaging.taskiq_broker import taskiq_broker
23 from src.repositories.order_repository import OrderRepository
24 from src.repositories.product_repository import ProductRepository
25 from src.repositories.user_repository import UserRepository
26 from src.repositories.report_repository import ReportRepository
27 from src.services.cache_service import CacheService
28 from src.services.order_service import OrderService
29 from src.services.product_service import ProductService
30 from src.services.user_service import UserService
31 from src.services.report_service import ReportService
32
33 logger = logging.getLogger(__name__)
34
35
36 class DatabaseConfig:
37     """Конфигурация базы данных."""
38
```

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

```
src > main.py
111     async def provide_order_repository(db_session: AsyncSession) -> OrderRepository:
112         """Провайдер репозитория заказов."""
113         return OrderRepository(db_session)
114
115
116     async def provide_report_repository(db_session: AsyncSession) -> ReportRepository:
117         """Провайдер репозитория отчетов."""
118         return ReportRepository(db_session)
119
120
121     async def provide_user_service(
122         user_repository: UserRepository, cache_service: CacheService
123     ) -> UserService:
124         """Провайдер сервиса пользователей."""
125         return UserService(user_repository, cache_service)
126
127
128     async def provide_product_service(
129         product_repository: ProductRepository, cache_service: CacheService
130     ) -> ProductService:
131         """Провайдер сервиса продуктов."""
132         return ProductService(product_repository, cache_service)
133
134
135     async def provide_order_service(
136         order_repository: OrderRepository,
137         product_repository: ProductRepository,
138         user_repository: UserRepository,
139     ) -> OrderService:
140         """Провайдер сервиса заказов."""
141         return OrderService(order_repository, product_repository, user_repository)
142
143
144     async def provide_report_service(
145         report_repository: ReportRepository,
146     ) -> ReportService:
147         """Провайдер сервиса отчетов."""
148         return ReportService(report_repository)
149
150
```

## 2. Добавлены провайдеры зависимостей:

- provide\_report\_repository() – провайдер репозитория отчетов
- provide\_report\_service() - провайдер сервиса отчетов

### Описание:

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

```
src > main.py
214     async def lifespan(app: Litestar):
252         finally:
276             except Exception:
278
279
280     def create_app() -> Litestar:
281         """Фабрика для создания приложения Litestar."""
282         return Litestar(
283             route_handlers=[
284                 UserController,
285                 ProductController,
286                 OrderController,
287                 ReportController,
288             ],
289             dependencies={
290                 # Database
291                 "db_session": Provide(provide_db_session),
292                 # Cache
293                 "cache_service": Provide(provide_cache_service),
294                 # Repositories
295                 "user_repository": Provide(provide_user_repository),
296                 "product_repository": Provide(provide_product_repository),
297                 "order_repository": Provide(provide_order_repository),
298                 "report_repository": Provide(provide_report_repository),
299                 # Services
300                 "user_service": Provide(provide_user_service),
301                 "product_service": Provide(provide_product_service),
302                 "order_service": Provide(provide_order_service),
303                 "report_service": Provide(provide_report_service),
304             },
305             lifespan=lifespan,
306             debug=True,
307         )
308
309
310     app = create_app()
311
```

### 3. Регистрация в приложении:

- ReportController добавлен в список route\_handlers - так приложение узнаёт о новых REST API эндпоинтах (/report).
- Провайдеры report\_repository и report\_service зарегистрированы в общем словаре dependencies для глобального использования.

```
File Edit Selection View Go Run Terminal Help < > Q Lab8

EXPLORER
LAB8
src
controllers
product_controller.py
report_controller.py
user_controller.py
messaging
__pycache__
tasks
__pycache__
_init_.py
report.py
__init__.py
broker.py
order.py
product.py
taskiq_broker.py
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
> OUTLINE

main.py x
src > main.py
212
213     @asynccontextmanager
214     async def lifespan(app: Litestar):
215         """Управление жизненным циклом приложения."""
216         global redis_client
217
218         redis_client = redis_config.create_client()
219         try:
220             await redis_client.ping()
221             logger.info("Redis connection established")
222         except Exception as e:
223             logger.error(f"Redis connection failed: {e}")
224             raise
225
226     try:
227         from src.messaging import order, product # noqa: F401
228         from src.messaging.tasks import report # noqa: F401
229         logger.info("Message handlers and tasks imported successfully")
230     except Exception as e:
231         logger.error(f"Failed to import handlers/tasks: {e}")
232         raise
233
234     shutdown_event = asyncio.Event()
235
236     faststream_task = asyncio.create_task(_faststream_broker_connect(shutdown_event))
237
238     start_taskiq_in_this_process = os.getenv("START_TASKIQ_WORKER", "0") == "1"
239
240     taskiq_task = None
241     if start_taskiq_in_this_process:
242         logger.info("START_TASKIQ_WORKER=1 -> starting Taskiq worker in this process")
243         taskiq_task = asyncio.create_task(_taskiq_worker_run(shutdown_event))
244     else:
245         logger.info(
246             "START_TASKIQ_WORKER not set (or !=1). "
247             "Taskiq worker will NOT be started in this process."
248         )
249
```

```
File Edit Selection View Go Run Terminal Help < > Q Lab8

EXPLORER
LAB8
src
controllers
product_controller.py
report_controller.py
user_controller.py
messaging
__pycache__
tasks
__pycache__
_init_.py
report.py
__init__.py
broker.py
order.py
product.py
taskiq_broker.py
models
repositories
schemas
services
utils
__init__.py
Dockerfile
entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
> OUTLINE

main.py x
src > main.py
214     async def lifespan(app: Litestar):
241         if start_taskiq_in_this_process:
242             else:
243                 logger.info(
244                     "START_TASKIQ_WORKER not set (or !=1). "
245                     "Taskiq worker will NOT be started in this process."
246                 )
247
248     try:
249         yield
250     finally:
251         logger.info("Starting shutdown sequence...")
252         shutdown_event.set()
253
254         for task, name in [(faststream_task, "FastStream"), (taskiq_task, "Taskiq")]:
255             if task is None:
256                 continue
257             task.cancel()
258             try:
259                 await task
260             except asyncio.CancelledError:
261                 logger.info(f"{name} task cancelled")
262             except Exception:
263                 logger.exception(f"Error in {name} task")
264
265         try:
266             await redis_client.close()
267             logger.info("Redis connection closed")
268         except Exception:
269             logger.exception("Error closing Redis")
270
271         try:
272             await engine.dispose()
273             logger.info("Database engine disposed")
274         except Exception:
275             logger.exception("Error disposing engine")
276
277
```

#### 4. Импорт задач в lifespan:

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

После этих изменений приложение становится целостным:

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

#### Демонстрация работы системы

```
app-1      | Waiting for PostgreSQL to be ready...
taskiq-scheduler-1 | Waiting for PostgreSQL to be ready...

taskiq-worker-1    | Waiting for PostgreSQL to be ready...

taskiq-scheduler-1 | PostgreSQL is up
app-1             | PostgreSQL is up
taskiq-worker-1   | PostgreSQL is up

taskiq-scheduler-1 | Skipping migrations (SKIP_MIGRATIONS=1)
taskiq-worker-1   | Skipping migrations (SKIP_MIGRATIONS=1)
app-1            | Running migrations...

taskiq-scheduler-1 | Starting application...Enable Watch
```

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

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

```

app-1 | INFO [alembic.runtime.migration] Context impl PostgresqlImpl.
app-1 | INFO [alembic.runtime.migration] Will assume transactional DDL.
app-1 | Starting application...
app-1 | INFO: Started server process [17]
app-1 | INFO: Waiting for application startup.
app-1 | INFO - 2025-12-16 18:12:54,109 - main - main - Redis connection established

rabbitmq-1 | 2025-12-16 18:12:54.131179+00:00 [info] <0.774.0> accepting AMQP connection <0.774.0> (172.19.0.5:57990 -> 172.19.0.4:5672)

app-1 | INFO - 2025-12-16 18:12:54,118 - main - main - Message handlers and tasks imported successfully

app-1 | INFO - 2025-12-16 18:12:54,118 - main - main - START_TASKIQ_WORKER not set (or !=1). Taskiq worker will NOT be started in this process.
app-1 | INFO - 2025-12-16 18:12:54,118 - main - main - Starting FastStream broker (attempt: 1/10)...
INFO: uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
2025-12-16 18:12:54,152 INFO - | order | - 'SubscribeOrder' waiting for messages
2025-12-16 18:12:54,160 INFO - | product | - 'SubscribeProduct' waiting for messages
INFO - 2025-12-16 18:12:54,167 - main - main - FastStream broker started successfully

```

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

```

taskiq-worker-1 | [2025-12-16 18:13:17,333][taskiq.worker][INFO ]|[MainProcess] Pid of a main process: 12
taskiq-worker-1 | [2025-12-16 18:13:17,333][taskiq.worker][INFO ]|[MainProcess] Starting 2 worker processes.
taskiq-worker-1 | [2025-12-16 18:13:17,346][taskiq.process-manager][INFO ]|[MainProcess] Started process worker-0 with pid 13
taskiq-worker-1 | [2025-12-16 18:13:17,360][taskiq.process-manager][INFO ]|[MainProcess] Started process worker-1 with pid 14
taskiq-worker-1 | INFO - 2025-12-16 18:13:20,221 - taskiq.worker - utils - Importing tasks from module src.messaging.tasks.report

taskiq-worker-1 | INFO - 2025-12-16 18:13:20,221 - taskiq.worker - utils - Importing tasks from module src.messaging.tasks.report

```

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

```

src > main.py
214     async def lifespan(app: litestar):
215         if start_taskiq_in_this_process:
216             else:
217                 logger.info(
218                     "START_TASKIQ_WORKER not set (or !=1). "
219                     "Taskiq worker will NOT be started in this process."
220                 )
221
222             try:
223                 yield
224             finally:
225                 logger.info("Starting shutdown cleanup...")
226
227             rabbitmq-1 | 2025-12-16 18:13:28.259839+00:00 [info] <0.852.0> connection <0.852.0> (172.19.0.8:49268 -> 172.19.0.4:5672): user 'guest' authenticated and granted access to vhost '/'
228             rabbitmq-1 | 2025-12-16 18:13:28.264594+00:00 [info] <0.855.0> connection <0.855.0> (172.19.0.8:49278 -> 172.19.0.4:5672): user 'guest' authenticated and granted access to vhost '/'
229             rabbitmq-1 | 2025-12-16 18:13:28.277880+00:00 [info] <0.878.0> accepting AMQP connection <0.878.0> (172.19.0.8:49282 -> 172.19.0.4:5672)
230
231             taskiq-worker-1 | INFO - 2025-12-16 18:13:20.335 - taskiq.receiver.receiver - receiver - Listening started.
232             rabbitmq-1 | 2025-12-16 18:13:20.284783+00:00 [info] <0.881.0> accepting AMQP connection <0.881.0> (172.19.0.8:49292 -> 172.19.0.4:5672)
233
234             taskiq-worker-1 | INFO - 2025-12-16 18:13:20.339 - taskiq.receiver.receiver - receiver - Listening started.
235             taskiq-scheduler-1 | INFO - 2025-12-16 18:13:20.370 - taskiq.worker - utils - Importing tasks from module src.messaging.tasks.report
236
237             rabbitmq-1 | 2025-12-16 18:13:20.2886940+00:00 [info] <0.878.0> connection <0.878.0> (172.19.0.8:49282 -> 172.19.0.4:5672): user 'guest' authenticated and granted access to vhost '/'
238             taskiq-scheduler-1 | INFO - 2025-12-16 18:13:20.380 - taskiq.cli.scheduler.run - run - Starting scheduler.
239             rabbitmq-1 | 2025-12-16 18:13:20.294870+00:00 [info] <0.881.0> connection <0.881.0> (172.19.0.8:49292 -> 172.19.0.4:5672): user 'guest' authenticated and granted access to vhost '/'
240             taskiq-scheduler-1 | INFO - 2025-12-16 18:13:20.443 - taskiq.cli.scheduler.run - run - Startup completed.
241             rabbitmq-1 | 2025-12-16 18:13:20.303428+00:00 [warning] <0.869.0> Deprecated Features: 'transient_nonexcl_queues': Feature 'transient_nonexcl_queues' is deprecated.
242             rabbitmq-1 | 2025-12-16 18:13:20.303428+00:00 [warning] <0.869.0> By default, this feature can still be used for now.
243             rabbitmq-1 | 2025-12-16 18:13:20.303428+00:00 [warning] <0.869.0> Its use will not be permitted by default in a future minor RabbitMQ version and the f

```

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

```

src > main.py
214     async def lifespan(app: litestar):
215         taskiq_task = asyncio.create_task(taskiq_worker_run(shutdown_event))
216
217             rabbitmq-1 | 2025-12-16 18:13:20.400618+00:00 [info] <0.933.0> connection <0.933.0> (172.19.0.6:33088 -> 172.19.0.4:5672): user 'guest' authenticated and granted access to vhost '/'
218             taskiq-scheduler-1 | INFO - 2025-12-16 18:13:21.002 - taskiq.cli.scheduler.run - run - Sending task src.messaging.tasks.report:generate_test_report with schedule_id generate_test_report.
219             taskiq-worker-1 | INFO - 2025-12-16 18:13:21.011 - taskiq.receiver.receiver - receiver - Executing task src.messaging.tasks.report:generate_test_report with ID: dab32724b0b45e79be9c25958be7
220             taskiq-worker-1 | INFO - 2025-12-16 18:13:21.013 - src.messaging.tasks.report - report - [TEST] Generating report for 2025-12-16
221             taskiq-worker-1 | 2025-12-16 18:13:21.121 INFO sqlalchemy.engine.Engine select pg_catalog.version()
222             taskiq-worker-1 | 2025-12-16 18:13:21.121 INFO sqlalchemy.engine.Engine [raw sql] ()
223             taskiq-worker-1 | 2025-12-16 18:13:21.121 - sqlalchemy.Engine - base - select pg_catalog.version()
224             taskiq-worker-1 | INFO - 2025-12-16 18:13:21.121 - sqlalchemy.Engine - base - [raw sql]()
225             taskiq-worker-1 | 2025-12-16 18:13:21.128 INFO sqlalchemy.engine.Engine select current_schema()
226             taskiq-worker-1 | 2025-12-16 18:13:21.128 INFO sqlalchemy.engine.Engine [raw sql]()
227             taskiq-worker-1 | 2025-12-16 18:13:21.128 - sqlalchemy.Engine - base - select current_schema()
228             taskiq-worker-1 | 2025-12-16 18:13:21.128 - sqlalchemy.Engine - base - [raw sql]()
229             taskiq-worker-1 | 2025-12-16 18:13:21.134 INFO sqlalchemy.engine.Engine show standard_conforming_strings
230             taskiq-worker-1 | 2025-12-16 18:13:21.135 INFO sqlalchemy.engine.Engine [raw sql]()
231             taskiq-worker-1 | 2025-12-16 18:13:21.134 - sqlalchemy.Engine - base - show standard_conforming_strings
232             taskiq-worker-1 | 2025-12-16 18:13:21.141 INFO sqlalchemy.engine.Engine BEGIN (implicit)
233             taskiq-worker-1 | 2025-12-16 18:13:21.141 - sqlalchemy.Engine - base - BEGIN (implicit)
234             taskiq-worker-1 | 2025-12-16 18:13:21.176 INFO sqlalchemy.engine.Engine SELECT order_reports.report_at, order_reports.order_id, order_reports.count_product, order_reports.total_amount
235             taskiq-worker-1 | FROM order_reports
236             taskiq-worker-1 | WHERE order_reports.report_at = $1::DATE
237             taskiq-worker-1 | 2025-12-16 18:13:21.176 - sqlalchemy.Engine - base - [generated in 0.00085s] (datetime.date(2025, 12, 16),)
238             taskiq-worker-1 | INFO - 2025-12-16 18:13:21.209 - src.messaging.tasks.report - report - [TEST] Report for 2025-12-16: 0 orders, 0 products, $0 total value
239             taskiq-worker-1 | 2025-12-16 18:13:21.210 INFO sqlalchemy.engine.Engine ROLLBACK
240             taskiq-worker-1 | INFO - 2025-12-16 18:13:21.210 - sqlalchemy.Engine - base - ROLLBACK
241             rabbitmq-1 | 2025-12-16 18:14:12.405001+00:00 [notice] <0.86.0> alarm_handler: (set,system memory high watermark,[])
242             taskiq-scheduler-1 | INFO - 2025-12-16 18:14:21.001 - taskiq.cli.scheduler.run - run - Sending task src.messaging.tasks.report:generate_test_report with schedule_id generate_test_report.
243             taskiq-worker-1 | INFO - 2025-12-16 18:14:21.014 - taskiq.receiver.receiver - receiver - Executing task src.messaging.tasks.report:generate_test_report with

```

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

```

src > main.py
214     async def lifespan(app: litestar):
243         taskiq_task = asyncio.create_task(_taskiq_worker_run(shutdown_event))

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]}}}

Тестируем операции 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}}


3. Тестирование операций с заказами

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

====

Тестируемение завершено

```

- Запустили добавление данных через очереди из 6 лабы

```

curl -X 'GET' \
  'http://localhost:8000/report?date=2025-12-16' \
  -H 'accept: application/json'

Request URL
http://localhost:8000/report?date=2025-12-16

Server response

Code Details
200 Response body
[{"report_at": "2025-12-16", "order_id": 4, "count_product": 2, "total_amount": 1359.97}, {"report_at": "2025-12-16", "order_id": 5, "count_product": 2, "total_amount": 299.96000000000004}, {"report_at": "2025-12-16", "order_id": 6, "count_product": 2, "total_amount": 259.96999999999997}]

Response headers
content-length: 266
content-type: application/json
date: Tue, 16 Dec 2025 18:19:48 GMT
server: unicorn

```

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

```
Curl
curl -X 'GET' \
'http://localhost:8000/report/summary?date=2025-12-16' \
-H 'accept: application/json'

Request URL
http://localhost:8000/report/summary?date=2025-12-16

Server response
Code Details
200 Response body
{
    "date": "2025-12-16",
    "total_orders": 3,
    "total_products": 6,
    "total_value": 1919.9
}
Response headers
content-length: 78
content-type: application/json
date: Tue, 16 Dec 2025 18:21:05 GMT
server: uvicorn

Responses
```

- Сводный отчет

## Ответы на вопросы

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

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

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

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

## 2. Что делает Планировщик (Scheduler)?

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

Основные функции:

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

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

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

**Проблемы:**

- Накопление задач в очереди - каждые 5 секунд планировщик добавляет

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

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

### Решения:

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

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

**cron\_offset** вы укажете? В чем подвох, если сервер работает по времени UTC?

**Cron-выражение:**

```
python@taskiq_broker.task(
```

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

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

**Подвохи при работе сервера по времени UTC:**

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

**Лучшая практика:**

- Хранить в БД и настройках UTC

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

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

Масштабирование при резком росте задач:

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

**Что произойдёт при отказе одного воркера:**

- 1) Сообщения не теряются: если воркер упал до подтверждения (ack), брокер вернёт сообщение в очередь — другой воркер его подхватит.
- 2) Задержка выполнения: задача выполнится позже, когда другой воркер

взьмёт сообщение.

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

Меры защиты:

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

**При отказе планировщика (scheduler):**

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

**Вывод:**

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

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