Лабораторная работа 7. Основы работы с Redis

Жунёв Андрей Александрович РИМ-150950

Цель работы:

овладеть базовыми навыками установки, подключения и взаимодействия с Redis в python. Изучить основные структуры данных Redis и их применение на практике.

Ход работы:

1. Добавление контейнера Redis в Docker Compose

Добавляем сервиса Redis в docker-compose.yml

```
▷ Run Service
redis:
  image: redis:7-alpine
  container_name: redis
  ports:
    - "6379:6379"
  volumes:
    - redis-data:/data
  networks:
    - pg_network_lab2
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 5s
    timeout: 3s
    retries: 5
  restart: unless-stopped

▷ Run Service
```

Обновляю зависимости в разделе app

```
      - REDIS_HOST=${REDIS_HOST:-redis}
      - REDIS_PORT=${REDIS_PORT:-6379}
      - REDIS_DB=${REDIS_DB:-0}
      - REDIS_DECODE_RESPONSES=${REDIS_DECODE_RESPONSES:-true}
    ports:
      - "8000:8000"
    volumes:
      - ./app:/app/app
      - ./main.py:/app/main.py
      - ./producer.py:/app/producer.py
    networks:
      - pg_network_lab2
    depends_on:
      db:
        condition: service_healthy
      rabbitmq:
        condition: service_started
      redis:
        condition: service_healthy
    command: ["uv", "run", "python", "main.py"]
```

2. Установка Redis

Добавляем зависимости в pyproject.toml, затем синхронизируем, чтобы пакетный менеджер (uv в моем случае подтянул нужные зависимости)

```
dependencies = [
    "alembic>=1.17.0",
    "asyncpg>=0.30.0",
    "faststream[rabbit]>=0.5.0",
    "litestar>=2.18.0",
    "pika>=1.3.0",
    "psycopg2-binary>=2.9.11",
    "pydantic[email]>=2.12.4",
    "python-dotenv>=1.1.1",
    "redis>=5.0.0",
```

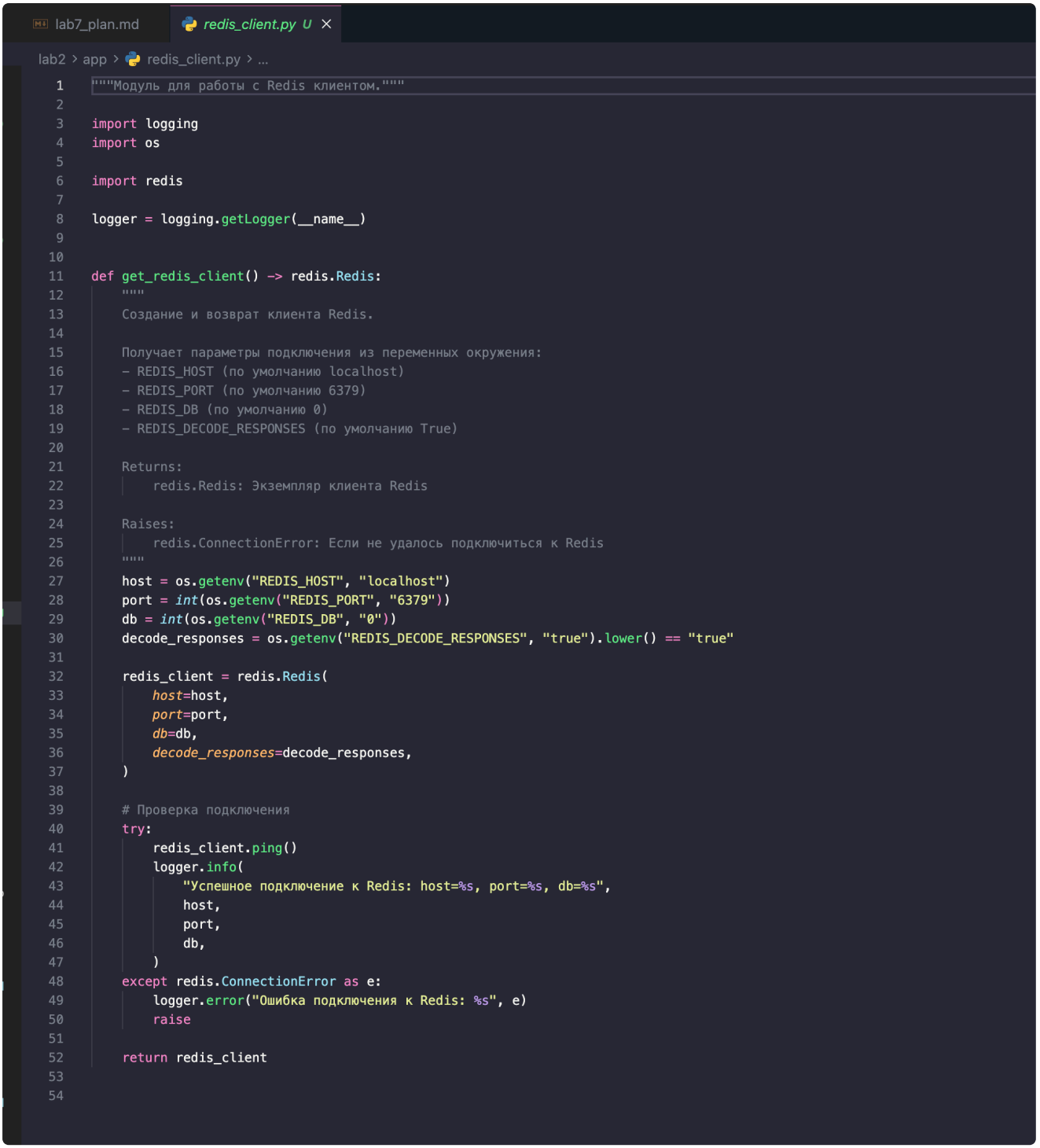
```
Problems  Output  Debug Console  Terminal  Ports

→  lab2 git:(main) ✗ uv sync
Resolved 75 packages in 5ms
Audited 73 packages in 1ms
→  lab2 git:(main) ✗
```

3. Создание модуля для работы с Redis

Создание модуля для работы с Redis необходимо для централизованного управления подключением и операциями с кэшем. Отдельный модуль обеспечивает переиспользование кода, упрощает тестирование и позволяет легко изменять конфигурацию подключения. Использование dependency injection гарантирует правильное управление жизненным циклом подключения.

Реализована функция подключения в созданом модуле для работы с



lab7_plan.md | redis_client.py U

lab2 > app > redis_client.py > ...

```python
"""Модуль для работы с Redis клиентом."""

import logging
import os

import redis

logger = logging.getLogger(__name__)


def get_redis_client() -> redis.Redis:
    """
    Создание и возврат клиента Redis.

    Получает параметры подключения из переменных окружения:
    - REDIS_HOST (по умолчанию localhost)
    - REDIS_PORT (по умолчанию 6379)
    - REDIS_DB (по умолчанию 0)
    - REDIS_DECODE_RESPONSES (по умолчанию True)

    Returns:
        redis.Redis: Экземпляр клиента Redis

    Raises:
        redis.ConnectionError: Если не удалось подключиться к Redis
    """
    host = os.getenv("REDIS_HOST", "localhost")
    port = int(os.getenv("REDIS_PORT", "6379"))
    db = int(os.getenv("REDIS_DB", "0"))
    decode_responses = os.getenv("REDIS_DECODE_RESPONSES", "true").lower() == "true"

    redis_client = redis.Redis(
        host=host,
        port=port,
        db=db,
        decode_responses=decode_responses,
    )

    # Проверка подключения
    try:
        redis_client.ping()
        logger.info(
            "Успешное подключение к Redis: host=%s, port=%s, db=%s",
            host,
            port,
            db,
        )
    except redis.ConnectionError as e:
        logger.error("Ошибка подключения к Redis: %s", e)
        raise

    return redis_client


```

Реализована функция проверки подключения

```python

def ping_redis(redis_client: redis.Redis) -> bool:
    """
    Проверка доступности Redis.

    Args:
        redis_client: Клиент Redis для проверки

    Returns:
        bool: True при успешном подключении, False при ошибке
    """
    try:
        redis_client.ping()
        logger.info("Проверка подключения к Redis: успешно")
        return True
    except redis.ConnectionError as e:
        logger.warning("Ошибка проверки подключения к Redis: %s", e)
        return False


```

Проведена интеграция с dependency injection

```python

def provide_redis_client() -> redis.Redis:
    """
    Провайдер клиента Redis.

    Создает и возвращает клиент Redis с использованием настроек из переменных окружения.
    Клиент создается при каждом запросе (stateless).

    Returns:
        redis.Redis: Экземпляр клиента Redis
    """
    return get_redis_client()

```

4. Знакомство с основными структурами данных Redis

Ознакомление с основными структурами данных Redis необходимо для понимания возможностей системы кэширования. Redis поддерживает различные типы данных: строки, списки, множества, хэши и упорядоченные множества. Каждый тип данных имеет свои особенности и применение, что позволяет выбрать оптимальную структуру для конкретной задачи кэширования.

Для ознакомления создан тестовый файл с примерами работы redis с разными типами данных:

строки

```python

def test_strings(client: redis.Redis) -> None:
    """Тестирование работы со строками (Strings) в Redis."""
    print("\n=== Тестирование Strings (строки) ===")

    # Установка и получение значения
    print("\n1. Установка и получение значения:")
    client.set("user:name", "Иван")
    name = client.get("user:name")
    print(f"   Установлено: user:name = 'Иван'")
    print(f"   Получено: {name}")
    # При decode_responses=True не нужен decode('utf-8')

    # Установка значения с TTL (Time To Live)
    print("\n2. Установка значения с TTL (1 час = 3600 секунд):")
    client.setex("session:123", 3600, "active")
    ttl = client.ttl("session:123")
    print(f"   Установлено: session:123 = 'active' с TTL 3600 секунд")
    print(f"   Оставшееся время жизни: {ttl} секунд")

    # Проверка существования ключа
    print("\n3. Проверка существования ключа:")
    exists = client.exists("user:name")
    print(f"   Ключ 'user:name' существует: {exists}")

    # Работа с числами - счетчик
    print("\n4. Работа с числами (счетчик):")
    client.set("counter", 0)
    print(f"   Начальное значение счетчика: {client.get('counter')}")

    # Увеличение на 1
    client.incr("counter")
    print(f"   После incr (увеличить на 1): {client.get('counter')}")

    # Увеличение на указанное значение
    client.incrby("counter", 5)
    print(f"   После incrby('counter', 5) (увеличить на 5): {client.get('counter')}")

```

списки

```python
def test_lists(client: redis.Redis) -> None:
    """Тестирование работы со списками (Lists) в Redis."""
    print("\n=== Тестирование Lists (списки) ===")

    # Добавление элементов в начало списка (lpush - left push)
    print("\n1. Добавление элементов в начало списка (lpush):")
    client.lpush("tasks", "task1", "task2")
    print("   Выполнено: lpush('tasks', 'task1', 'task2')")
    print("   lpush добавляет элементы в начало списка (слева)")
    tasks = client.lrange("tasks", 0, -1)
    print(f"   Текущее состояние списка: {tasks}")

    # Добавление элементов в конец списка (rpush - right push)
    print("\n2. Добавление элементов в конец списка (rpush):")
    client.rpush("tasks", "task3", "task4")
    print("   Выполнено: rpush('tasks', 'task3', 'task4')")
    print("   rpush добавляет элементы в конец списка (справа)")
    tasks = client.lrange("tasks", 0, -1)
    print(f"   Текущее состояние списка: {tasks}")

    # Получение всех элементов списка
    print("\n3. Получение всех элементов списка:")
    all_tasks = client.lrange("tasks", 0, -1)
    print(f"   lrange('tasks', 0, -1): {all_tasks}")
    print("   Параметры: 0 - начало списка, -1 - конец списка")

```

множества

```python
def test_sets(client: redis.Redis) -> None:
    """Тестирование работы с множествами (Sets) в Redis."""
    print("\n=== Тестирование Sets (множества) ===")

    # Добавление элементов в множество
    print("\n1. Добавление элементов в множество:")
    client.sadd("tags", "python", "redis", "database")
    print("   Выполнено: sadd('tags', 'python', 'redis', 'database')")
    print("   Множества хранят уникальные элементы (без дубликатов)")

    client.sadd("languages", "python", "java", "javascript")
    print("   Выполнено: sadd('languages', 'python', 'java', 'javascript')")

    # Получение всех элементов множества
    print("\n2. Получение всех элементов множества:")
    all_tags = client.smembers("tags")
    print(f"   smembers('tags'): {all_tags}")

    # Проверка принадлежности элемента
    print("\n3. Проверка принадлежности элемента:")
    is_member = client.sismember("tags", "python")
    print(f"   sismember('tags', 'python'): {is_member}")
    print("   Возвращает True, если элемент принадлежит множеству, иначе False")
```

хэши

```python
def test_hashes(client: redis.Redis) -> None:
    """Тестирование работы с хэшами (Hashes) в Redis."""
    print("\n=== Тестирование Hashes (хэши) ===")

    # Установка нескольких полей в хэше
    print("\n1. Установка нескольких полей в хэше:")
    client.hset("user:1000", mapping={"name": "Иван", "age": "30", "city": "Москва"})
    print("   Выполнено: hset('user:1000', mapping={'name': 'Иван', 'age': '30', 'city': 'Москва'})")
    print("   Хэши удобны для хранения объектов с несколькими полями")

    # Установка одного поля в хэше
    print("\n2. Установка одного поля в хэше:")
    client.hset("user:1000", "email", "ivan@example.com")
    print("   Выполнено: hset('user:1000', 'email', 'ivan@example.com')")

    # Получение значения поля
    print("\n3. Получение значения поля:")
    name = client.hget("user:1000", "name")
    print(f"   hget('user:1000', 'name'): {name}")

    age = client.hget("user:1000", "age")
    print(f"   hget('user:1000', 'age'): {age}")
```

упорядоченные множества

```python
def test_sorted_sets(client: redis.Redis) -> None:
    """Тестирование работы с упорядоченными множествами (Sorted Sets) в Redis."""
    print("\n=== Тестирование Sorted Sets (упорядоченные множества) ===")

    # Добавление элементов с оценками (scores)
    print("\n1. Добавление элементов с оценками (scores):")
    client.zadd("leaderboard", {"player1": 100, "player2": 200, "player3": 150})
    print("   Выполнено: zadd('leaderboard', {'player1': 100, 'player2': 200, 'player3': 150})")
    print("   Упорядоченные множества хранят элементы с числовыми оценками для сортировки")

    # Получение элементов по индексу (от начала до конца)
    print("\n2. Получение элементов по индексу:")
    all_players = client.zrange("leaderboard", 0, -1)
    print(f"   zrange('leaderboard', 0, -1): {all_players}")
    print("   Возвращает элементы в порядке возрастания оценки (от меньшей к большей)")

    # Получение первых N элементов с оценками
    print("\n3. Получение первых N элементов с оценками:")
    top_players = client.zrange("leaderboard", 0, 2, withscores=True)
    print(f"   zrange('leaderboard', 0, 2, withscores=True): {top_players}")
    print("   Параметр withscores=True возвращает элементы вместе с их оценками")
```

работа с TTL

```python

def test_ttl(client: redis.Redis) -> None:
    """Тестирование работы с TTL (Time To Live) в Redis."""
    print("\n=== Тестирование TTL (Time To Live) ===")

    # Установка значения с TTL
    print("\n1. Установка значения с TTL (1 час = 3600 секунд):")
    client.setex("session:123", 3600, "active")
    print("   Выполнено: setex('session:123', 3600, 'active')")
    print("   setex устанавливает значение и TTL одновременно (атомарная операция)")

    # Проверка TTL
    ttl = client.ttl("session:123")
    print(f"   ttl('session:123'): {ttl} секунд")
    print("   Возвращает оставшееся время жизни ключа в секундах")

    # Установка TTL для существующего ключа
    print("\n2. Установка TTL для существующего ключа:")
    client.set("key", "value")
    print("   Создан ключ без TTL")
    initial_ttl = client.ttl("key")
    print(f"   ttl('key') до установки TTL: {initial_ttl}")
    print("   -1 означает, что TTL не установлен (ключ постоянный)")

```

По итогу создан тестовый файл, все тесты выполняются успешно на поднятом контейнере Redis.

```
================================================================================
                    СВОДКА РЕЗУЛЬТАТОВ ТЕСТИРОВАНИЯ
================================================================================

Время выполнения: 0.02 секунд
Дата и время: 2025-12-08 18:30:22

--------------------------------------------------------------------------------
Тип данных                                Статус              Детали
--------------------------------------------------------------------------------
Strings (строки)                          ✓ Успешно           -
Lists (списки)                            ✓ Успешно           -
Sets (множества)                          ✓ Успешно           -
Hashes (хэши)                             ✓ Успешно           -
Sorted Sets (упорядоченные множества)     ✓ Успешно           -
TTL (Time To Live)                        ✓ Успешно           -
--------------------------------------------------------------------------------

Всего тестов: 6
Успешно: 6 ✓
Ошибок: 0 ✗

================================================================================
                          ВСЕ ТЕСТЫ ПРОЙДЕНЫ УСПЕШНО!
================================================================================

=== Тестирование завершено ===
(lab2) ➜  lab2 git:(main) ✗
```

5. Добавление кэширования для пользователя

Реализация кэширования для пользователей необходима для ускорения доступа к часто запрашиваемым данным. Кэширование данных пользователя на 1 час (3600 секунд) позволяет значительно снизить нагрузку на базу данных при повторных запросах. При обновлении данных пользователя необходимо удалять соответствующие данные из кэша (инвалидация), чтобы обеспечить актуальность данных при следующем запросе.

Реализация функции кэширования

```python
def get_user_from_cache(
    redis_client: redis.Redis, user_id: int
) -> dict | None:
    """
    Получение данных пользователя из кэша Redis.

    Args:
        redis_client: Клиент Redis для выполнения операций
        user_id: Идентификатор пользователя

    Returns:
        dict | None: Словарь с данными пользователя, если найден в кэше,
                     иначе None
    """
    key = f"user:{user_id}"

    try:
        cached_data = redis_client.get(key)
        if cached_data is None:
            logger.debug("Cache miss для пользователя: user_id=%s", user_id)
            return None

        user_data = json.loads(cached_data)
        logger.info("Cache hit для пользователя: user_id=%s", user_id)
        return user_data

    except redis.ConnectionError as e:
        logger.warning(
            "Ошибка подключения к Redis при получении пользователя из кэша: %s",
            e,
        )
        return None
    except json.JSONDecodeError as e:
        logger.error(
            "Ошибка десериализации данных пользователя из кэша: user_id=%s, error=%s",
            user_id,
            e,
        )
        # Удаляем поврежденные данные из кэша
        try:
            redis_client.delete(key)
        except redis.ConnectionError:
            pass
        return None
```

Реализация функции сохранения в кэш

```python

def set_user_to_cache(
    redis_client: redis.Redis,
    user_id: int,
    user_data: dict,
    ttl: int = 3600,
) -> None:
    """
    Сохранение данных пользователя в кэш Redis с TTL.

    Args:
        redis_client: Клиент Redis для выполнения операций
        user_id: Идентификатор пользователя
        user_data: Словарь с данными пользователя для сохранения
        ttl: Время жизни ключа в секундах (по умолчанию 1 час = 3600 секунд)
    """
    key = f"user:{user_id}"

    try:
        json_data = json.dumps(user_data)
        redis_client.setex(key, ttl, json_data)
        logger.info(
            "Данные пользователя сохранены в кэш: user_id=%s, ttl=%s секунд",
            user_id,
            ttl,
        )
    except redis.ConnectionError as e:
        logger.warning(
            "Ошибка подключения к Redis при сохранении пользователя в кэш: %s",
            e,
        )
        # Не выбрасываем исключение, чтобы не блокировать основную логику
    except (TypeError, ValueError) as e:
        logger.error(
            "Ошибка сериализации данных пользователя для кэша: user_id=%s, error=%s",
            user_id,
            e,
        )
        # Не выбрасываем исключение, чтобы не блокировать основную логику


```

Функция удаления из кэша

```python
def delete_user_from_cache(redis_client: redis.Redis, user_id: int) -> None:
    """
    Удаление данных пользователя из кэша Redis.

    Args:
        redis_client: Клиент Redis для выполнения операций
        user_id: Идентификатор пользователя
    """
    key = f"user:{user_id}"

    try:
        deleted = redis_client.delete(key)
        if deleted:
            logger.info(
                "Данные пользователя удалены из кэша: user_id=%s", user_id
            )
        else:
            logger.debug(
                "Ключ пользователя не найден в кэше: user_id=%s", user_id
            )
    except redis.ConnectionError as e:
        logger.warning(
            "Ошибка подключения к Redis при удалении пользователя из кэша: %s",
            e,
        )
        # Не выбрасываем исключение, чтобы не блокировать основную логику


```

Интеграция кэширования в UserService

```python
class UserService:
    def __init__(
        self.redis_client = redis_client

    async def get_by_id(self, session: AsyncSession, user_id: int) -> User | None:
        """
        Получить пользователя по ID с использованием кэширования.
        Args:
            session: Асинхронная сессия базы данных
            user_id: ID пользователя (int)

        Returns:
            User объект или None, если не найден
        """
        # Попытка получить данные из кэша
        if self.redis_client:
            cached_data = get_user_from_cache(self.redis_client, user_id)
            if cached_data is not None:
                # Преобразуем словарь обратно в объект User
                # Преобразуем строки ISO формата обратно в datetime
                created_at = (
                    datetime.fromisoformat(cached_data["created_at"])
                    if isinstance(cached_data["created_at"], str)
                    else cached_data["created_at"]
                )
                updated_at = (
                    datetime.fromisoformat(cached_data["updated_at"])
                    if cached_data.get("updated_at")
                    and isinstance(cached_data["updated_at"], str)
                    else cached_data.get("updated_at")
                )
                return User(
                    id=cached_data["id"],
                    username=cached_data["username"],
                    email=cached_data["email"],
                    description=cached_data.get("description"),
                    created_at=created_at,
                    updated_at=updated_at,
                )

```

Обновление dependency injection

```python
async def provide_user_service(
    user_repository: UserRepository, redis_client: redis.Redis
) -> UserService:
    """
    Провайдер сервиса пользователей.

    Args:
        user_repository: Репозиторий пользователей (внедряется через DI)
        redis_client: Клиент Redis для кэширования (внедряется через DI)

    Returns:
        UserService: Экземпляр сервиса пользователей
    """
    return UserService(user_repository, redis_client)

```

6. Модуль для кэширования продукции

Реализация кэширования для продукции необходима для ускорения доступа к данным о товарах. Кэширование данных продукции с ограничением в **10 минут (600 секунд)** позволяет снизить нагрузку на базу данных при частых запросах каталога товаров. При обновлении данных продукции необходимо **обновлять данные в кэше**, чтобы обеспечить актуальность данных без необходимости повторного запроса к БД.

Функция кэширования

```python
def get_product_from_cache(
    redis_client: redis.Redis, product_id: int
) -> dict | None:
    """
    Получение данных продукции из кэша Redis.

    Args:
        redis_client: Клиент Redis для выполнения операций
        product_id: Идентификатор продукции

    Returns:
        dict | None: Словарь с данными продукции, если найден в кэше,
                     иначе None
    """
    key = f"product:{product_id}"

    try:
        cached_data = redis_client.get(key)
        if cached_data is None:
            logger.debug("Cache miss для продукции: product_id=%s", product_id)
            return None

        product_data = json.loads(cached_data)
        logger.info("Cache hit для продукции: product_id=%s", product_id)
        return product_data

    except redis.ConnectionError as e:
        logger.warning(
            "Ошибка подключения к Redis при получении продукции из кэша: %s",
            e,
        )
        return None
    except json.JSONDecodeError as e:
        logger.error(
            "Ошибка десериализации данных продукции из кэша: product_id=%s, error=%s",
            product_id,
            e,
        )
        # Удаляем поврежденные данные из кэша
        try:
            redis_client.delete(key)
        except redis.ConnectionError:
            pass
        return None


```

Функция сохранения в кэш

```python
def set_product_to_cache(
    redis_client: redis.Redis,
    product_id: int,
    product_data: dict,
    ttl: int = 600,
) -> None:
    """
    Сохранение данных продукции в кэш Redis с TTL.

    Args:
        redis_client: Клиент Redis для выполнения операций
        product_id: Идентификатор продукции
        product_data: Словарь с данными продукции для сохранения
        ttl: Время жизни ключа в секундах (по умолчанию 10 минут = 600 секунд)
    """
    key = f"product:{product_id}"

    try:
        json_data = json.dumps(product_data)
        redis_client.setex(key, ttl, json_data)
        logger.info(
            "Данные продукции сохранены в кэш: product_id=%s, ttl=%s секунд",
            product_id,
            ttl,
        )
    except redis.ConnectionError as e:
        logger.warning(
            "Ошибка подключения к Redis при сохранении продукции в кэш: %s",
            e,
        )
        # Не выбрасываем исключение, чтобы не блокировать основную логику
    except (TypeError, ValueError) as e:
        logger.error(
            "Ошибка сериализации данных продукции для кэша: product_id=%s, error=%s",
            product_id,
            e,
        )
        # Не выбрасываем исключение, чтобы не блокировать основную логику

```

Функция обновления кэша

```python
def update_product_in_cache(
    redis_client: redis.Redis,
    product_id: int,
    product_data: dict,
    ttl: int = 600,
) -> None:
    """
    Обновление данных продукции в кэше Redis с TTL.

    При обновлении продукции данные обновляются в кэше (в отличие от пользователей,
    где происходит инвалидация). Это обеспечивает актуальность данных без
    необходимости повторного запроса к БД.

    Args:
        redis_client: Клиент Redis для выполнения операций
        product_id: Идентификатор продукции
        product_data: Словарь с обновленными данными продукции
        ttl: Время жизни ключа в секундах (по умолчанию 10 минут = 600 секунд)
    """
    key = f"product:{product_id}"

    try:
        json_data = json.dumps(product_data)
        # setex работает как set, если ключа нет - он будет создан
        redis_client.setex(key, ttl, json_data)
        logger.info(
            "Данные продукции обновлены в кэше: product_id=%s, ttl=%s секунд",
            product_id,
            ttl,
        )
    except redis.ConnectionError as e:
        logger.warning(
            "Ошибка подключения к Redis при обновлении продукции в кэше: %s",
            e,
        )
        # Не выбрасываем исключение, чтобы не блокировать основную логику
    except (TypeError, ValueError) as e:
        logger.error(
            "Ошибка сериализации данных продукции для обновления кэша: product_id=%s, error=%s",
            product_id,
            e,
        )
        # Не выбрасываем исключение, чтобы не блокировать основную логику


```

Функция удаления кэша

```python
def delete_product_from_cache(redis_client: redis.Redis, product_id: int) -> None:
    """
    Удаление данных продукции из кэша Redis.

    Args:
        redis_client: Клиент Redis для выполнения операций
        product_id: Идентификатор продукции
    """
    key = f"product:{product_id}"

    try:
        deleted = redis_client.delete(key)
        if deleted:
            logger.info(
                "Данные продукции удалены из кэша: product_id=%s", product_id
            )
        else:
            logger.debug(
                "Ключ продукции не найден в кэше: product_id=%s", product_id
            )
    except redis.ConnectionError as e:
        logger.warning(
            "Ошибка подключения к Redis при удалении продукции из кэша: %s",
            e,
        )
        # Не выбрасываем исключение, чтобы не блокировать основную логику


```

В итоге,р еализовано кэширование данных продукции на 10 минут, при обновлении продукции данные обновляются в кэше, кэширование интегрировано в ProductService

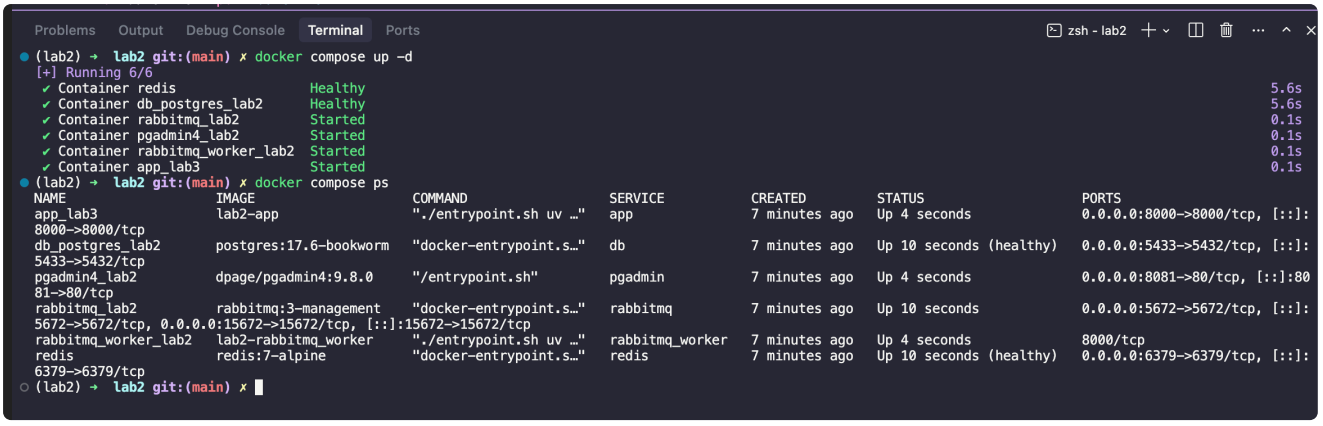
7. Проведение тестирования

Для проведения тестирования создан скрипт со следующими тестами:

1. Тестирование кэширования пользователей:
   - Тест 1: Получение пользователя (cache miss) — проверка сохранения в кэш и TTL 3600 секунд
   - Тест 2: Получение пользователя (cache hit) — проверка скорости ответа из кэша
   - Тест 3: Обновление пользователя (инвалидация кэша) — проверка удаления ключа и получения обновленных данных
2. Тестирование кэширования продукции:
   - Тест 1: Получение продукции (cache miss) — проверка сохранения в кэш и TTL 600 секунд
   - Тест 2: Получение продукции (cache hit) — проверка скорости ответа из кэша
   - Тест 3: Обновление продукции (обновление кэша) — проверка обновления данных в кэше и TTL
3. Тестирование TTL:

- Проверка TTL для пользователей (3600 секунд)
- Проверка TTL для продукции (600 секунд)

4. Тестирование производительности:
   - Сравнение времени ответа с кэшем и без кэша



Docker compose успешно развернут, запускаем тестирование.

Все тесты успешно пройдены

Обновил README.md - добавил раздел о Redis, обновил раздел Docker, добавил информацию о структуре кэша и примеры использования Redis CLI.

Изменения закомичены, добавлен тег lab_7. Исправил ошибки, которые отметил pylint при первой попытке коммита - слишком обобщено были написаны исключения в try в нескольких проверках.

```
Your code has been rated at 9.94/10 (previous run: 10.00/10, -0.06)

[INFO] Restored changes from /Users/2madeira/.cache/pre-commit/patch1765203406-12026.
(lab2) ➜  lab2 git:(main) ✗ git add .
(lab2) ➜  lab2 git:(main) ✗ git commit -m "feat(redis): добавлен Redis, обновлен Docker-compose, реализовано кэшир
тесты"
[WARNING] Unstaged files detected.
[INFO] Stashing unstaged files to /Users/2madeira/.cache/pre-commit/patch1765203554-12732.
Black....................................................................Passed
isort....................................................................Passed
pylint...................................................................Passed
[INFO] Restored changes from /Users/2madeira/.cache/pre-commit/patch1765203554-12732.
[main fa7150f] feat(redis): добавлен Redis, обновлен Docker-compose, реализовано кэширования для пользователей и п
 21 files changed, 1828 insertions(+), 18 deletions(-)
 create mode 100644 lab2/app/__pycache__/redis_client.cpython-313.pyc
 create mode 100644 lab2/app/cache/__init__.py
 create mode 100644 lab2/app/cache/__pycache__/__init__.cpython-313.pyc
 create mode 100644 lab2/app/cache/__pycache__/product_cache.cpython-313.pyc
 create mode 100644 lab2/app/cache/__pycache__/user_cache.cpython-313.pyc
 create mode 100644 lab2/app/cache/product_cache.py
 create mode 100644 lab2/app/cache/user_cache.py
 create mode 100644 lab2/app/redis_client.py
 create mode 100644 lab2/test_redis_cache.py
 create mode 100644 lab2/test_redis_types.py
(lab2) ➜  lab2 git:(main) ✗ git tag lab_7
(lab2) ➜  lab2 git:(main) ✗ git push origin lab_7
Enumerating objects: 45, done.
Counting objects: 100% (45/45), done.
Delta compression using up to 10 threads
Compressing objects: 100% (30/30), done.
Writing objects: 100% (30/30), 38.08 KiB | 7.62 MiB/s, done.
Total 30 (delta 16), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (16/16), completed with 13 local objects.
To https://github.com/2made1ra/App_Dev_sem_1.git
 * [new tag]         lab_7 -> lab_7
(lab2) ➜  lab2 git:(main) ✗
```

Вопросы

1. В чем заключается основное преимущество хранения данных в оперативной памяти (in-memory) по сравнению с дисковыми БД?

Основное преимущество хранения данных в оперативной памяти заключается в скорости доступа к данным. Оперативная память обеспечивает доступ к данным в наносекундах, в то время как дисковые операции требуют миллисекунд или даже секунд из-за механических ограничений жестких дисков (поиск, вращение, чтение). Это делает Redis в десятки и сотни раз быстрее традиционных дисковых баз данных для операций чтения и записи.

Кроме того, in-memory хранилища идеально подходят для кэширования часто запрашиваемых данных, счетчиков, сессий и других временных данных, где скорость важнее персистентности. Однако стоит отметить, что данные в оперативной памяти теряются при перезагрузке сервера, поэтому Redis поддерживает различные механизмы персистентности (RDB, AOF) для критически важных данных.

2. Для чего нужен параметр `decode_responses=True` при создании клиента Redis?

Параметр `decode_responses=True` автоматически декодирует ответы Redis из байтовых строк в обычные строки Python. По умолчанию Redis возвращает данные в виде байтовых строк (bytes), что требует ручного декодирования через `.decode('utf-8')` при каждом получении данных. С включенным `decode_responses=True` клиент автоматически преобразует все ответы в строки, что упрощает работу с данными.

Это особенно удобно при работе с текстовыми данными, JSON и другими строковыми форматами, так как избавляет от необходимости постоянно вызывать методы декодирования. Однако для работы с бинарными данными этот параметр следует отключить, чтобы сохранить исходный формат данных.

3. Что такое TTL (Time To Live) ключа и как он используется в Redis?

TTL (Time To Live) — это время жизни ключа в Redis, выраженное в секундах. После истечения TTL ключ автоматически удаляется из базы данных. TTL устанавливается при создании ключа (например, через команду `SETEX`) или для существующего ключа через команду `EXPIRE`. Проверить оставшееся время жизни можно командой `TTL`, которая возвращает количество секунд до истечения, -1 если TTL не установлен, или -2 если ключ не существует.

TTL широко используется для кэширования данных с ограниченным временем актуальности, управления сессиями пользователей, временных токенов и других данных, которые должны автоматически удаляться через определенное время. Это позволяет не засорять базу данных устаревшими данными и автоматически обновлять кэш при истечении срока действия.

4. Объясните разницу между командами `r.lpush()` и `r.rpush()` для списков.

Команды `lpush()` и `rpush()` добавляют элементы в список Redis, но в разные его концы. `lpush()` (left push) добавляет элементы в начало списка (слева), а `rpush()` (right push) добавляет элементы в конец списка (справа). Это позволяет использовать списки Redis как стеки (LIFO - Last In First Out) с помощью `lpush`/`lpop` или как очереди (FIFO - First In First Out) с помощью `lpush`/`rpop` или `rpush`/`lpop`.

Например, если выполнить `lpush("list", "a", "b")`, а затем `rpush("list", "c", "d")`, список будет содержать элементы в порядке `["b", "a", "c", "d"]` (при чтении слева направо). Это различие критично для реализации различных структур данных и алгоритмов, таких как очереди задач, истории операций или стеки вызовов.

5. Как обеспечить атомарность операций в Redis?

Атомарность операций в Redis обеспечивается тем, что Redis является однопоточным сервером, который обрабатывает команды последовательно. Каждая

команда выполняется полностью до начала следующей, что гарантирует отсутствие race conditions при выполнении одной команды. Для сложных операций, требующих нескольких команд, Redis предоставляет механизмы транзакций через команды `MULTI`, `EXEC` и `WATCH`, которые позволяют выполнить группу команд атомарно.

Кроме того, многие команды Redis сами по себе атомарны и выполняют несколько операций за один вызов. Например, `SETEX` атомарно устанавливает значение и TTL, `INCR` атомарно увеличивает число, а `LPUSH` и `RPUSH` атомарно добавляют элементы в список. Для более сложных сценариев можно использовать Lua скрипты, которые выполняются атомарно на стороне сервера.

6. Как в Redis реализована репликация и кластеризация?

Репликация в Redis реализована через механизм master-slave (в Redis 5.0+ называется master-replica). Один сервер Redis работает как master (принимает записи), а один или несколько серверов работают как replicas (получают копии данных). Репликация асинхронная: master отправляет команды записи репликам, которые выполняют их для поддержания синхронизации. Настройка выполняется через команду `REPLICAOF` или параметр конфигурации `replicaof`.

Кластеризация Redis использует распределенную архитектуру, где данные разделяются между несколькими узлами через механизм хэш-слотов (16384 слотов). Каждый ключ хэшируется и назначается определенному слоту, который обслуживается одним из узлов кластера. Клиенты автоматически перенаправляются на нужный узел при выполнении команд. Кластер обеспечивает высокую доступность через репликацию каждого узла и автоматическое переключение при отказе master-узла. Для работы кластера требуется минимум 3 master-узла.