

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 №7:
Основы работы с Redis**

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

Цель работы

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

Задачи

1. Добавить контейнер Redis в docker-compose и настроить подключение к Redis из Python-приложения.
2. Изучить работу с основными структурами данных Redis: строки (установка, получение, TTL), списки (добавление, получение элементов), множества, хэши, упорядоченные множества.
3. Реализовать кэширование для двух сущностей (пользователи и продукты) с разным TTL и инвалидацией кэша.
4. Запустить и проверить работоспособность.

Часть 1: Добавление контейнера Redis с базовой конфигурацией

Файл: *docker-compose.yml*

EXPLORER

- LAB7
 - src
 - repositories
 - product_repository.py
 - user_repository.py
 - schemas
 - _pycache_
 - __init__.py
 - order.py
 - product.py
 - user.py
 - services
 - _pycache_
 - __init__.py
 - cache_service.py
 - order_service.py
 - product_service.py
 - user_service.py
 - utils
 - __init__.py
 - Dockerfile
 - entrypoint.sh
 - main.py
 - tests
 - .dockerignore
 - .gitignore
 - .python-version
 - alembic.ini
 - docker-compose.yml
 - pyproject.toml
 - README.md
 - redis_example.py
 - test_system.py
- OUTLINE
- TIMELINE

docker-compose.yml

```

version: '3.8'

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

```

EXPLORER

- LAB7
 - src
 - repositories
 - product_repository.py
 - user_repository.py
 - schemas
 - _pycache_
 - __init__.py
 - order.py
 - product.py
 - user.py
 - services
 - _pycache_
 - __init__.py
 - cache_service.py
 - order_service.py
 - product_service.py
 - user_service.py
 - utils
 - __init__.py
 - Dockerfile
 - entrypoint.sh
 - main.py
 - tests
 - .dockerignore
 - .gitignore
 - .python-version
 - alembic.ini
 - docker-compose.yml
 - pyproject.toml
 - README.md
 - redis_example.py
 - test_system.py
- OUTLINE
- TIMELINE

docker-compose.yml

```

version: '3.8'

services:
  db:
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user -d db"]
      interval: 10s
      timeout: 5s
      retries: 5

  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

```

The screenshot shows a code editor interface with two main panes. The left pane, titled 'EXPLORER', displays the project structure of 'LAB7'. It includes a 'src' folder containing 'repositories', 'schemas', 'services', 'utils', 'tests', and configuration files like '.dockerignore', '.gitignore', '.python-version', and 'alembic.ini'. The 'docker-compose.yml' file is highlighted in blue at the bottom of this list. The right pane shows the content of 'docker-compose.yml'. The configuration defines two services: 'rabbitmq' and 'app'. The 'rabbitmq' service depends on 'db' and has a healthcheck defined. The 'app' service builds from the current directory, uses a Dockerfile in 'src', and runs on port 8000. It depends on 'db' and 'rabbitmq', and also depends on 'redis'. It uses the 'app_network' network and mounts volumes for 'postgres_data', 'rabbitmq_data', and 'redis-data'.

```

version: '3.8'

services:
  rabbitmq:
    depends_on:
      - db
    healthcheck:
      test: ["CMD", "rabbitmq-diagnostics", "ping"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 40s

  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
    ports:
      - "8000:8000"
    depends_on:
      - db
      - rabbitmq
      - redis
    networks:
      - app_network

volumes:
  postgres_data:
  rabbitmq_data:
  redis-data:
networks:
  app_network:

```

Ход выполнения:

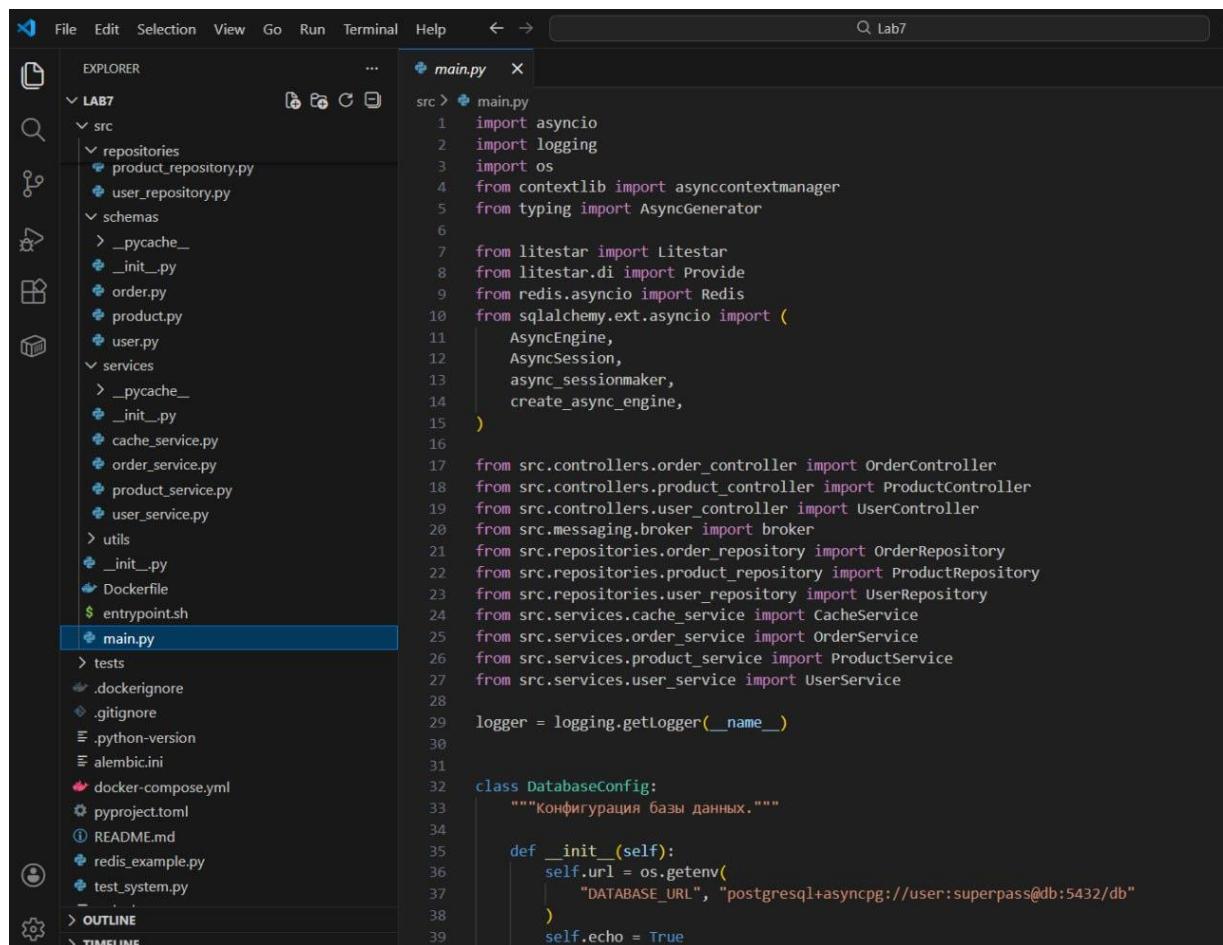
Был добавлен сервис Redis с полной конфигурацией:

- image: redis:7-alpine - использовали облегченную версию Redis 7
- ports: "6379:6379" - стандартный порт Redis
- command: redis-server --appendonly yes - включили персистентность данных, чтобы данные сохранялись на диск (по умолчанию Redis хранит данные только в памяти)
- volumes: redis-data:/data - монтирование volume для сохранения данных между перезапусками контейнера (без volume данные теряются при удалении контейнера)
- healthcheck - проверка здоровья контейнера через команду redis-cli ping, что обеспечивает правильную последовательность запуска сервисов

- networks: app_network - подключение к общей сети приложения для взаимодействия с другими сервисами

Часть 2: Подключение к Redis

Файл: *main.py*



The screenshot shows a code editor interface with a dark theme. On the left is the Explorer sidebar showing a project structure for 'LAB7'. The 'src' directory contains 'repositories' (with 'product_repository.py' and 'user_repository.py'), 'schemas' (with '_pycache_'), 'services' (with '_pycache_'), and 'utils' (with '_init_.py'). The 'tests' directory contains 'Dockerfile' and 'entrypoint.sh'. In the center, the main.py file is open in the editor. The code imports various modules from 'src' and 'src.services'. It defines a 'DatabaseConfig' class with an __init__ method that sets the database URL to 'DATABASE_URL' (defaulting to 'postgresql+asyncpg://user:superpass@db:5432/db') and enables echo. It also imports 'logging' and creates a logger.

```

File Edit Selection View Go Run Terminal Help ← → Q Lab7

EXPLORER
LAB7
src
  repositories
    product_repository.py
    user_repository.py
  schemas
    _pycache_
    __init__.py
  services
    _pycache_
    __init__.py
    cache_service.py
    order_service.py
    product_service.py
    user_service.py
  utils
    __init__.py
  Dockerfile
  entrypoint.sh
main.py
tests
.dockerignore
.gitignore
.python-version
alembic.ini
.docker-compose.yml
pyproject.toml
README.md
redis_example.py
test_system.py
OUTLINE
TIMELINE

main.py

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.messaging.broker import broker
21 from src.repositories.order_repository import OrderRepository
22 from src.repositories.product_repository import ProductRepository
23 from src.repositories.user_repository import UserRepository
24 from src.services.cache_service import CacheService
25 from src.services.order_service import OrderService
26 from src.services.product_service import ProductService
27 from src.services.user_service import UserService
28
29 logger = logging.getLogger(__name__)
30
31
32 class DatabaseConfig:
33     """Конфигурация базы данных."""
34
35     def __init__(self):
36         self.url = os.getenv(
37             "DATABASE_URL", "postgresql+asyncpg://user:superpass@db:5432/db"
38         )
39         self.echo = True

```

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

- File Explorer (Left):** Shows the project structure under "LAB7".
 - src:** Contains "repositories" (with "product_repository.py" and "user_repository.py"), "schemas" (with "order.py" and "product.py"), "services" (with "cache_service.py", "order_service.py", "product_service.py", "user_service.py", and "utils"), and "tests".
 - main.py:** The current file being edited.
- Search Bar:** Shows "Q Lab7".
- Code Editor (Right):** Displays the content of the "main.py" file.

```
src > main.py
32 class DatabaseConfig:
33     def __init__(self):
34
35         def create_engine(self) -> AsyncEngine:
36             """Создает engine для подключения к БД."""
37             return create_async_engine(
38                 self.url,
39                 echo=self.echo,
40                 pool_pre_ping=True,
41                 pool_size=10,
42                 max_overflow=20,
43             )
44
45
46
47
48
49
50
51
52 class RedisConfig:
53     """Конфигурация Redis."""
54
55     def __init__(self):
56         self.host="redis"
57         self.port=6379
58         self.db=0
59
60     def create_client(self) -> Redis:
61         """создает клиент Redis."""
62         return Redis(
63             host=self.host,
64             port=self.port,
65             db=self.db,
66             decode_responses=False,
67         )
68
69
70 db_config = DatabaseConfig()
71 engine = db_config.create_engine()
72 async_session_maker = async_sessionmaker(
73     engine,
74     class_=AsyncSession,
75     expire_on_commit=False,
76     autoflush=False,
```

The screenshot shows a code editor with a Python file named `main.py` open. The code defines several asynchronous session providers using the `async_sessionmaker` function from the `sqlalchemy.ext.asyncio` module. The `provide_db_session` provider returns an `AsyncSession`. The `provide_cache_service` provider returns a `CacheService` which uses a `redis_client`. The `provide_user_repository`, `provide_product_repository`, and `provide_order_repository` providers return instances of `UserRepository`, `ProductRepository`, and `OrderRepository` respectively, all utilizing the `db_session`. A tooltip for the `db_session` parameter in the `provide_db_session` definition is displayed, providing documentation for the provider's purpose and usage.

```
async_session_maker = async_sessionmaker(  
    ...  
)  
redis_config = RedisConfig()  
redis_client: Redis = None  
  
async def provide_db_session() -> AsyncGenerator[AsyncSession, None]:  
    """  
    Провайдер сессии базы данных.  
    Yields:  
        AsyncSession: Активная сессия БД  
    """  
    async with async_session_maker() as session:  
        try:  
            yield session  
        finally:  
            await session.close()  
  
async def provide_cache_service() -> CacheService:  
    """Провайдер сервиса кеширования."""  
    return CacheService(redis_client)  
  
async def provide_user_repository(db_session: AsyncSession) -> UserRepository:  
    """Провайдер репозитория пользователей."""  
    return UserRepository(db_session)  
  
async def provide_product_repository(db_session: AsyncSession) -> ProductRepository:  
    """Провайдер репозитория продуктов."""  
    return ProductRepository(db_session)  
  
async def provide_order_repository(db_session: AsyncSession) -> OrderRepository:  
    """Провайдер репозитория заказов."""  
    return OrderRepository(db_session)
```

File Edit Selection View Go Run Terminal Help

EXPLORER

- LAB7
 - src
 - repositories
 - product_repository.py
 - user_repository.py
 - schemas
 - _pycache_
 - __init__.py
 - order.py
 - product.py
 - user.py
 - services
 - _pycache_
 - __init__.py
 - cache_service.py
 - order_service.py
 - product_service.py
 - user_service.py
 - utils
 - __init__.py
 - Dockerfile
 - entrypoint.sh
 - main.py
 - tests
 - .dockerignore
 - .gitignore
 - .python-version
 - alembic.ini
 - docker-compose.yml
 - pyproject.toml
 - README.md
 - redis_example.py
 - test_system.py
- OUTLINE
- TIMELINE

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_user_service(
117         user_repository: UserRepository,
118         cache_service: CacheService
119     ) -> UserService:
120         """Провайдер сервиса пользователей."""
121         return UserService(user_repository, cache_service)
122
123
124     async def provide_product_service(
125         product_repository: ProductRepository,
126         cache_service: CacheService
127     ) -> ProductService:
128         """Провайдер сервиса продуктов."""
129         return ProductService(product_repository, cache_service)
130
131
132     async def provide_order_service(
133         order_repository: OrderRepository,
134         product_repository: ProductRepository,
135         user_repository: UserRepository,
136     ) -> OrderService:
137         """Провайдер сервиса заказов."""
138         return OrderService(order_repository, product_repository, user_repository)
139
140
141     async def _broker_connect_loop(shutdown_event: asyncio.Event) -> None:
142         """
143             Фоновая задача для запуска брокера и
144             поддержания его работы до завершения работы системы.
145             Брокер самостоятельно управляет переподключением
146             благодаря своему устойчивому соединению.
147         """
148
149         max_retries = 10
150         retry_delay = 5
151
152         for attempt in range(max_retries):
153             try:
154                 logger.info(
155                     f"Attempting to start Rabbit broker (attempt {attempt + 1}/{max_retries})..."
156                 )
157                 await broker.start()
158                 logger.info("Rabbit broker connected successfully")
159
160                 await shutdown_event.wait()
161
162                 logger.info("Shutdown requested, closing broker...")
163                 await broker.close()
164                 logger.info("Broker closed successfully")
165                 return
166
167             except Exception as e:
168                 logger.error(
169                     f"Broker connection error (attempt {attempt + 1}/{max_retries}): {e}"
170                 )
171             if attempt < max_retries - 1:
172                 logger.info(f"Retrying in {retry_delay} seconds...")
173                 await asyncio.sleep(retry_delay)
174             else:
175                 logger.error("Max retries reached, giving up on broker connection")
176                 raise
177
178

```

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

- File Explorer (Left):** Shows the project structure under the 'LAB7' folder. The 'main.py' file is selected and highlighted.
- Code Editor (Right):** Displays the content of the 'main.py' file. The code is written in Python and uses the `asyncio` library for asynchronous operations. It includes imports for `order` and `product` from the `src.messaging` module, and defines an `async def lifespan(app: Litestar)` function. The code handles Redis connection management and broker shutdown logic.
- Status Bar:** Shows the path 'src > main.py' and the line numbers 1/8 to 216.

```
src > main.py
1/8
179 @asynccontextmanager
180 async def lifespan(app: Litestar):
181     """
182     Управление жизненным циклом приложения.
183     """
184     Выполняется при старте и остановке приложения.
185     """
186     global redis_client
187
188     redis_client = redis_config.create_client()
189     try:
190         await redis_client.ping()
191         logger.info("Redis connection established")
192     except Exception as e:
193         logger.error(f"Redis connection failed: {e}")
194         raise
195
196     shutdown_event = asyncio.Event()
197     broker_task = asyncio.create_task(_broker_connect_loop(shutdown_event))
198
199     from src.messaging import order, product # noqa: F401
200
201     try:
202         yield
203     finally:
204         shutdown_event.set()
205
206         try:
207             await broker.close()
208         except Exception:
209             logger.exception("Error while closing broker")
210
211         broker_task.cancel()
212         try:
213             await broker_task
214         except asyncio.CancelledError:
215             pass
216         except Exception:
```

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

- File Explorer (Left):** Shows the project structure under the LAB7 folder:
 - src
 - repositories
 - product_repository.py
 - user_repository.py
 - schemas
 - _pycache_
 - __init__.py
 - order.py
 - product.py
 - user.py
 - services
 - _pycache_
 - __init__.py
 - cache_service.py
 - order_service.py
 - product_service.py
 - user_service.py
 - utils
 - __init__.py
 - Dockerfile
 - entrypoint.sh
 - Editor (Right):** The main.py file is open, showing Python code for a Litestar application.

```
async def lifespan(app: Litestar):
    finally:
        except asyncio.CancelledError:
            except Exception:
                logger.exception("Broker task raised while shutting down")

    try:
        await redis_client.close()
        logger.info("Redis connection closed")
    except Exception:
        logger.exception("Error closing Redis connection")

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

def create_app() -> Litestar:
    """
    Фабрика для создания приложения Litestar.

    Returns:
        Litestar: Сконфигурированное приложение
    """
    return Litestar(
        route_handlers=[
            UserController,
            ProductController,
            OrderController,
        ],
        dependencies={
            # Database
            "db_session": Provide(provide_db_session),
            # Cache
            "cache_service": Provide(provide_cache_service),
            # Repositories
            "user_repository": Provide(provide_user_repository),
            "product_repository": Provide(provide_product_repository),
        }
    )
```
 - Status Bar (Top):** Shows the current file as main.py and the title bar says "Q Lab7".

```
src > main.py
231 def create_app() -> Litestar:
232     return Litestar(
233         dependencies={
234             "cache_service": Provide(provide_cache_service),
235             # Repositories
236             "user_repository": Provide(provide_user_repository),
237             "product_repository": Provide(provide_product_repository),
238             "order_repository": Provide(provide_order_repository),
239             # Services
240             "user_service": Provide(provide_user_service),
241             "product_service": Provide(provide_product_service),
242             "order_service": Provide(provide_order_service),
243         },
244         lifespan=[lifespan],
245         debug=True,
246     )
247
248     app = create_app()
249
250     if __name__ == "__main__":
251         import uvicorn
252
253         uvicorn.run(
254             "main:app",
255             host="0.0.0.0",
256             port=8000,
257             log_level="info",
258         )
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
```

Ход выполнения:

1. Конфигурация Redis

- class RedisConfig- был добавлен класс конфигурации Redis
- host="redis" - имя сервиса из docker-compose для внутреннего обращения между контейнерами
- decode_responses=False - отключено автоматическое декодирование, т.к. данные будут храниться в JSON формате (при True все значения автоматически декодируются в строки!)
- redis.asyncio.Redis - асинхронный клиент используется вместо синхронного, не блокирует event loop при операциях с Redis

2. Инициализация Redis в жизненном цикле приложения

В функции lifespan была добавлена инициализация Redis:

- Создание единственного экземпляра Redis клиента при старте приложения: redis_client = redis_config.create_client()

- Проверка подключения через ping() для раннего обнаружения проблем
- Корректное закрытие соединения при остановке приложения для освобождения ресурсов

3. Провайдер для внедрения зависимостей

async def provide_cache_service() -> CacheService:

```
"""Провайдер сервиса кэширования."""
return CacheService(redis_client)
```

- Litestar вызывает provide_cache_service() при необходимости
- Функция создает экземпляр CacheService с глобальным redis_client
- Экземпляр внедряется в зависимые сервисы
- Каждый запрос получает доступ к кэшу через DI

Часть 3: Пример работы с Redis

Файл: *redis_example.py*

(Тестовый файл для демонстрации всех возможностей Redis, который можно запустить отдельно)

The screenshot shows a code editor interface with a dark theme. On the left is the Explorer sidebar showing a project structure under 'LAB7'. The 'src' folder contains several files: '_pycache_/_init_.py', 'order.py', 'product.py', 'user.py', 'services/_pycache_/_init_.py', 'cache_service.py', 'order_service.py', 'product_service.py', 'user_service.py', 'utils/_init_.py', 'Dockerfile', 'entrypoint.sh', 'main.py', 'tests/.dockerignore', '.gitignore', '.python-version', 'alembic.ini', 'docker-compose.yml', 'pyproject.toml', 'README.md', and 'redis_example.py'. The 'redis_example.py' file is selected and shown in the main editor area.

```
File Edit Selection View Go Run Terminal Help ← → Q Lab7

EXPLORER
LAB7
src
repositories
schemas
> _pycache_
> _init_.py
order.py
product.py
user.py
services
> _pycache_
> _init_.py
cache_service.py
order_service.py
product_service.py
user_service.py
utils
> _init_.py
Dockerfile
$ entrypoint.sh
main.py
> tests
.dockerignore
.gitignore
.python-version
alembic.ini
docker-compose.yml
pyproject.toml
README.md
redis_example.py
test_system.py
uv.lock
> OUTLINE
> TIMELINE

redis_example.py ×
redis_example.py
1 import redis
2
3 # Подключение к Redis
4 client = redis.Redis(host="localhost", port=6379, db=0, decode_responses=True)
5
6 # Проверка подключения
7 try:
8     client.ping()
9     print("Подключение к Redis успешно")
10 except redis.ConnectionError:
11     print("Ошибка подключения к Redis")
12     exit(1)
13
14 # Строки
15 client.set("user:name", "Иван")
16 name = client.get("user:name")
17 print(f"Строка user:name = {name}")
18
19 client.setex("session:123", 3600, "active")
20 print(f"Session TTL = {client.get('session:123')}")
21
22 # Счетчики
23 client.set("counter", 0)
24 client.incr("counter")
25 client.incrby("counter", 5)
26 val = client.decr("counter")
27 print(f"Counter после операций = {val}")
28
29 # Списки
30 client.delete("tasks")
31 client.lpush("tasks", "task1", "task2")
32 client.rpush("tasks", "task3", "task4")
33 tasks = client.lrange("tasks", 0, -1)
34 print(f"Список tasks = {tasks}")
35
36 first = client.lpop("tasks")
37 last = client.rpop("tasks")
38 print(f"Удалены: первый={first}, последний={last}, осталось={client.lrange('tasks', 0, -1)}")
```

The screenshot shows a code editor interface with a dark theme. The top navigation bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar labeled 'Lab7'. The left sidebar displays a file tree under 'LAB7' with various files like src, repositories, schemas, services, utils, tests, and configuration files. The main editor area has a title bar 'redis_example.py x'. The code itself is as follows:

```
34 print(f"Список tasks = {tasks}")
35
36 first = client.lpop("tasks")
37 last = client.rpop("tasks")
38 print(f"Удалены: первый={first}, последний={last}, осталось={client.lrange('tasks', 0, -1)}")
39
40 # Множества
41 client.delete("tags", "languages")
42 client.sadd("tags", "python", "redis", "database")
43 client.sadd("languages", "python", "java", "javascript")
44 print(f"Tags = {client.smembers('tags')}")
45 print(f"Пересечение tags и languages = {client.sinter('tags', 'languages')}")
46
47 # Хеш-таблицы
48 client.hset("user:1000", mapping={
49     "name": "Иван",
50     "age": "30",
51     "city": "Москва"
52 })
53 user_data = client.hgetall("user:1000")
54 print(f"User 1000 = {user_data}")
55
56 # Упорядоченные множества
57 client.delete("leaderboard")
58 client.zadd("leaderboard", {
59     "player1": 100,
60     "player2": 200,
61     "player3": 150
62 })
63 top = client.zrevrange("leaderboard", 0, 2, withscores=True)
64 print(f"Топ игроков = {top}")
65
66 print("Все операции выполнены успешно")
```

Файл содержит примеры использования всех основных структур данных Redis, чтобы запустить его отдельно для тестирования:

1. Строки - базовое хранилище ключ-значение
 2. TTL - автоматическое удаление ключей через время
 3. Счетчики - атомарные операции инкремента/декремента
 4. Списки - упорядоченные коллекции с быстрым доступом к концам
 5. Множества - уникальные элементы с операциями пересечения
 6. Хеш-таблицы - структурированное хранение объектов
 7. Упорядоченные множества - элементы с рейтингом для leaderboard

Пример запуска теста:

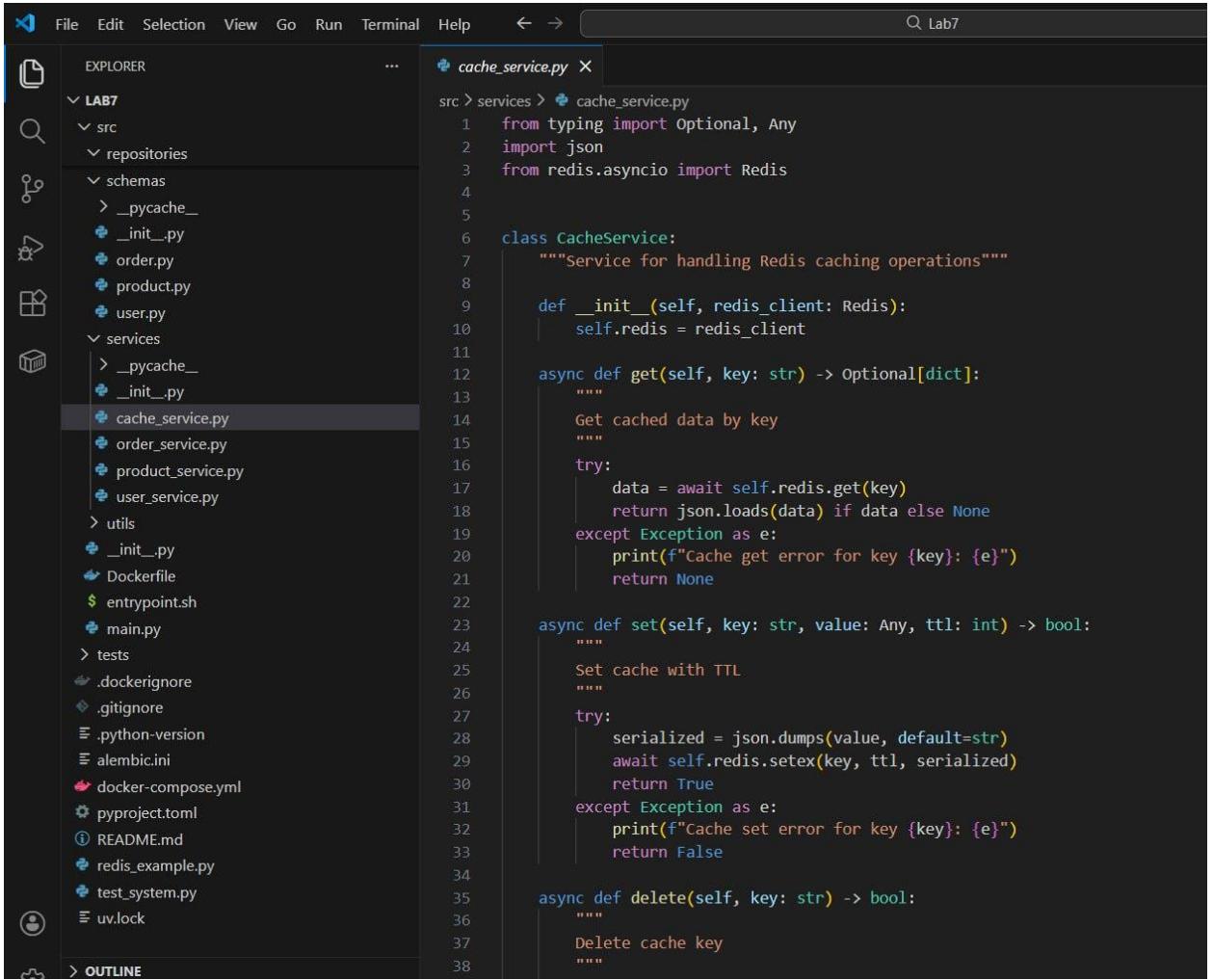
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

(lab7) PS C:\Users\Anastasia\Desktop\application-development-homework\Lab7> python .\redis_example.py
Подключение к Redis успешно
Строка user:name = Иван
Session с TTL = active
Counter после операций = 5
Список tasks = ['task2', 'task1', 'task3', 'task4']
Удалены: первый=task2, последний=task4, осталось=['task1', 'task3']
Tags = {'database', 'redis', 'python'}
Пересечение tags и languages = {'python'}
User 1000 = {'name': 'Иван', 'age': '30', 'city': 'Москва'}
Топ игроков = [('player2', 200.0), ('player3', 150.0), ('player1', 100.0)]
Все операции выполнены успешно
○ (lab7) PS C:\Users\Anastasia\Desktop\application-development-homework\Lab7>
```

Часть 4: Кэширование запросов и очищение запросов

Файл: *cache_service.py*

(Создание сервиса кэширования)



```
File Edit Selection View Go Run Terminal Help ← → Q Lab7

EXPLORER
LAB7
src
  repositories
    schemas
      __pycache__
      __init__.py
      order.py
      product.py
      user.py
    services
      __pycache__
      __init__.py
      cache_service.py
      order_service.py
      product_service.py
      user_service.py
      utils
        __init__.py
      Dockerfile
      entrypoint.sh
      main.py
    tests
    .dockerignore
    .gitignore
    .python-version
    alembic.ini
    docker-compose.yml
    pyproject.toml
    README.md
    redis_example.py
    test_system.py
    uv.lock

cache_service.py

cache_service.py
src > services > cache_service.py
1  from typing import Optional, Any
2  import json
3  from redis.asyncio import Redis
4
5
6  class CacheService:
7      """Service for handling Redis caching operations"""
8
9      def __init__(self, redis_client: Redis):
10         self.redis = redis_client
11
12     async def get(self, key: str) -> Optional[dict]:
13         """
14             Get cached data by key
15         """
16         try:
17             data = await self.redis.get(key)
18             return json.loads(data) if data else None
19         except Exception as e:
20             print(f"Cache get error for key {key}: {e}")
21             return None
22
23     async def set(self, key: str, value: Any, ttl: int) -> bool:
24         """
25             Set cache with TTL
26         """
27         try:
28             serialized = json.dumps(value, default=str)
29             await self.redis.setex(key, ttl, serialized)
30             return True
31         except Exception as e:
32             print(f"Cache set error for key {key}: {e}")
33             return False
34
35     async def delete(self, key: str) -> bool:
36         """
37             Delete cache key
38         """
```

```
cache_service.py
src > services > cache_service.py
6     class CacheService:
23         async def set(self, key: str, value: Any, ttl: int) -> bool:
31             except Exception as e:
32                 print(f"Cache set error for key {key}: {e}")
33                 return False
34
35         async def delete(self, key: str) -> bool:
36             """
37             Delete cache key
38             """
39
40             try:
41                 await self.redis.delete(key)
42                 return True
43             except Exception as e:
44                 print(f"Cache delete error for key {key}: {e}")
45                 return False
46
47         async def delete_pattern(self, pattern: str) -> int:
48             """
49             Delete all keys matching a pattern
50             """
51
52             try:
53                 keys = []
54                 async for key in self.redis.scan_iter(match=pattern):
55                     keys.append(key)
56
57                 if keys:
58                     return await self.redis.delete(*keys)
59                 return 0
60             except Exception as e:
61                 print(f"Cache delete pattern error for pattern {pattern}: {e}")
62                 return 0
```

Ход выполнения:

Была реализована централизованная логика работы с кэшем, а также автоматическая сериализация/десериализация JSON и обработка ошибок для отказоустойчивости.

Основные методы:

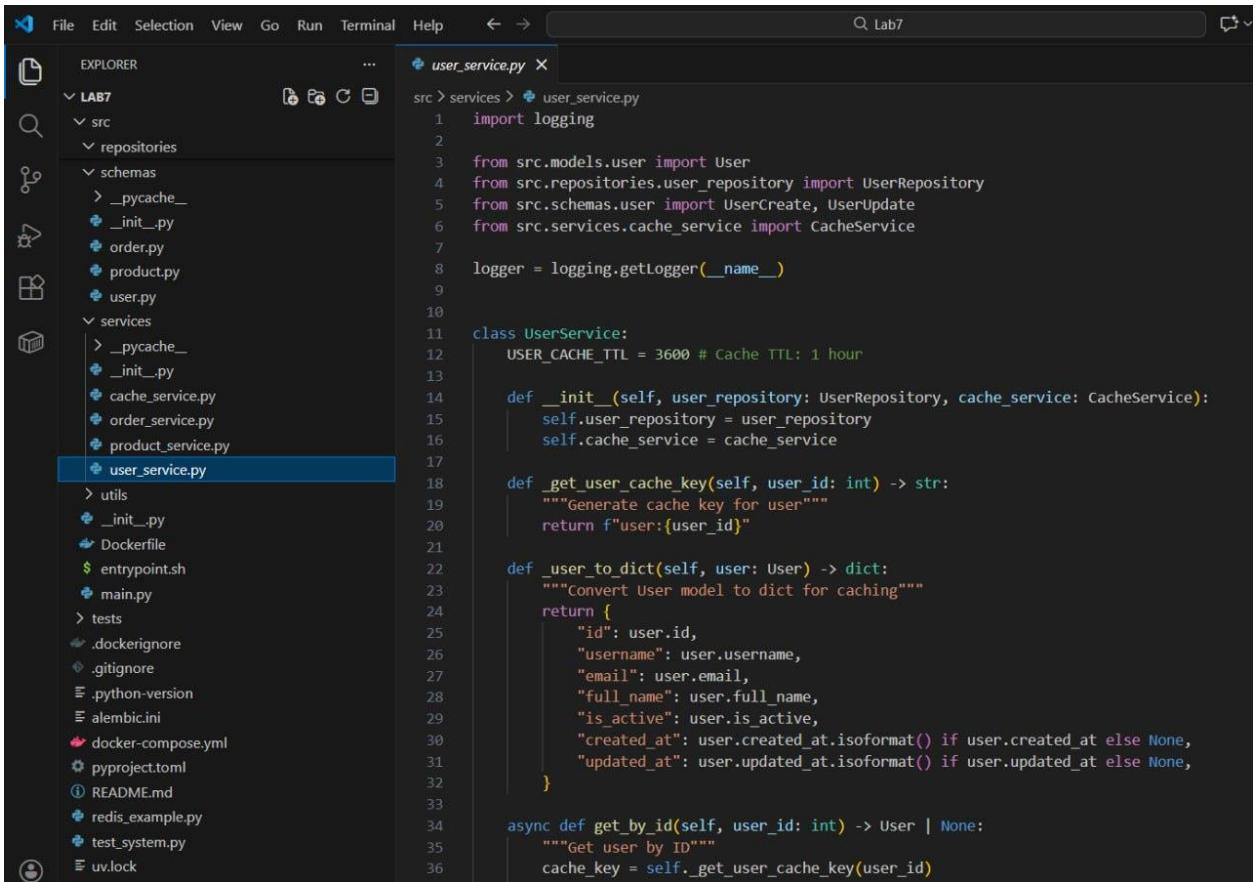
- **get(key)** - получение данных с десериализацией из JSON. Метод Получает bytes из Redis и десериализует JSON в Python dict. При ошибке возвращает None (приложение продолжит работу без кэша)
- **set(key, value, ttl)** - сохранение в кэш с автоматическим истечением.
 - json.dumps(value, default=str) - сериализует любые объекты
 - default=str обрабатывает datetime и другие не-JSON типы
 - setex(key, ttl, value) - устанавливает значение с автоматическим истечением
 - ttl в секундах: Redis сам удалит ключ через указанное время

- **delete(key)** - удаление одного ключа
 - **delete_pattern(pattern)** - массовое удаление по шаблону.
- scan_iter(match=pattern) - итератор по ключам, не блокирует Redis при большом количестве ключей.

Все методы обернуты в try-except для graceful degradation, поэтому если Redis недоступен, приложение продолжит работать, просто без кэширования (будет обращаться напрямую к БД).

Файл: *user_service.py*

(Кэширование пользователей (TTL: 1 час))



The screenshot shows the PyCharm IDE interface. On the left is the 'EXPLORER' sidebar displaying the project structure:

```

LAB
  src
    repositories
      schemas
        __pycache__
        __init__.py
        order.py
        product.py
        user.py
    services
      __pycache__
      __init__.py
      cache_service.py
      order_service.py
      product_service.py
      user_service.py
    utils
      __init__.py
      Dockerfile
      entrypoint.sh
      main.py
    tests
    .dockerignore
    .gitignore
    .python-version
    alembic.ini
    docker-compose.yml
    pyproject.toml
    README.md
    redis_example.py
    test_system.py
    uv.lock
  tests
  .gitignore
  .python-version
  alembic.ini
  docker-compose.yml
  pyproject.toml
  README.md
  redis_example.py
  test_system.py
  uv.lock

```

The 'user_service.py' file is selected in the sidebar and is open in the main editor window. The code implements a UserService class with methods for generating cache keys and converting User models to dictionaries. It also includes an asynchronous get_by_id method.

```

user_service.py
src > services > user_service.py
1 import logging
2
3 from src.models.user import User
4 from src.repositories.user_repository import UserRepository
5 from src.schemas.user import UserCreate, UserUpdate
6 from src.services.cache_service import CacheService
7
8 logger = logging.getLogger(__name__)
9
10 class UserService:
11     USER_CACHE_TTL = 3600 # Cache TTL: 1 hour
12
13     def __init__(self, user_repository: UserRepository, cache_service: CacheService):
14         self.user_repository = user_repository
15         self.cache_service = cache_service
16
17     def _get_user_cache_key(self, user_id: int) -> str:
18         """Generate cache key for user"""
19         return f"user:{user_id}"
20
21     def _user_to_dict(self, user: User) -> dict:
22         """Convert User model to dict for caching"""
23         return {
24             "id": user.id,
25             "username": user.username,
26             "email": user.email,
27             "full_name": user.full_name,
28             "is_active": user.is_active,
29             "created_at": user.created_at.isoformat() if user.created_at else None,
30             "updated_at": user.updated_at.isoformat() if user.updated_at else None,
31         }
32
33     async def get_by_id(self, user_id: int) -> User | None:
34         """Get user by ID"""
35         cache_key = self._get_user_cache_key(user_id)
36

```

```

File Edit Selection View Go Run Terminal Help < > Q Lab7

EXPLORER
LAB7
src
  repositories
    schemas
      __pycache__
      __init__.py
      order.py
      product.py
      user.py
  services
    __pycache__
    __init__.py
    cache_service.py
    order_service.py
    product_service.py
    user_service.py
    utils
      __init__.py
    Dockerfile
    entrypoint.sh
    main.py
  tests
    .dockerignore
    .gitignore
    .python-version
    alembic.ini
    docker-compose.yml
    pyproject.toml
    README.md
    redis_example.py
    test_system.py
    uv.lock

OUTLINE
services
  __pycache__
  __init__.py
  cache_service.py
  order_service.py
  product_service.py
  user_service.py
  utils
    __init__.py
  Dockerfile
  entrypoint.sh
  main.py
  tests
    .dockerignore
    .gitignore
    .python-version
    alembic.ini
    docker-compose.yml

user_service.py

src > services > user_service.py
11  class UserService:
34      async def get_by_id(self, user_id: int) -> User | None:
37          cached_data = await self.cache_service.get(cache_key)
38          if cached_data:
39              logger.info(f"Cache HIT for user {user_id}")
40              return cached_data
41
42          logger.info(f"Cache MISS for user {user_id}")
43          user = await self.user_repository.get_by_id(user_id)
44
45          if user:
46              user_dict = self._user_to_dict(user)
47              await self.cache_service.set(cache_key, user_dict, self.USER_CACHE_TTL)
48              return user
49
50
51      return None
52
53
54      async def get_by_filter(self, count: int, page: int, **kwargs) -> dict:
55          """Get a list of users with filtering and the total number"""
56          users = await self.user_repository.get_by_filter(count, page, **kwargs)
57          total = await self.user_repository.count(**kwargs)
58          return {"total": total, "items": users}
59
60      async def create(self, user_data: UserCreate) -> User:
61          """Create a new user"""
62          return await self.user_repository.create(user_data)
63
64      async def update(self, user_id: int, user_data: UserUpdate) -> User:
65          """Update user data"""
66          patch = (
67              user_data.model_dump(exclude_none=True)
68              if hasattr(user_data, "model_dump")
69              else dict(user_data)
70          )
71
72          updated_user = await self.user_repository.update(user_id, user_data)
73
74
75          updated_user = await self.user_repository.update(user_id, user_data)
76
77          if updated_user:
78              cache_key = self._get_user_cache_key(user_id)
79              user_dict = self._user_to_dict(updated_user)
80              await self.cache_service.set(cache_key, user_dict, self.USER_CACHE_TTL)
81              logger.info(f"Cache UPDATED for user {user_id}")
82
83          return updated_user
84
85      async def delete(self, user_id: int) -> None:
86          """Удалить пользователя"""
87          await self.user_repository.delete(user_id)
88
89          cache_key = self._get_user_cache_key(user_id)
90          await self.cache_service.delete(cache_key)
91          logger.info(f"Cache DELETED for user {user_id}")

```

Добавлено кэширование с требуемым временем жизни:

- USER_CACHE_TTL = 3600 (1 час = 3600 секунд)

Вспомогательные методы:

- get_user_cache_key() - генерация ключа кэша
- user_to_dict() - конвертация модели в словарь для кэширования.

Необходимо конвертировать в dict т.к. SQLAlchemy модели нельзя напрямую сериализовать в JSON, и таким образом мы можем контролировать, какие поля попадают в кэш.

- `get_by_id()` - стратегия кэширования при чтении. Используется Cache-Aside паттерн, сначала проверяем кэш, при промахе загружаем из БД. Добавлено логирование - отслеживание эффективности кэширования с помощью HIT/MISS.
 - `update()` - обновление кэша при изменении. При обновлении данных сразу обновляем кэш, чтобы избежать возврата устаревших данных.
 - `delete()` - удаление из кэша с инвалидацией - если не удалить из кэша, последующие запросы вернут удаленного пользователя и нарушится консистентность данных

Файл: *product_service.py* (Кэширование продуктов (TTL: 10 минут))

The screenshot shows a code editor interface with a navigation bar at the top. The left sidebar displays a file tree for a project named 'LAB7'. The 'src' directory contains 'repositories', 'schemas', and 'services'. Under 'services', there are files for 'cache_service.py', 'order_service.py', and 'product_service.py', which is currently selected and highlighted in blue. Other files in 'services' include 'user_service.py', 'utils', and 'Dockerfile'. The main workspace shows the content of 'product_service.py'. The code defines a class 'ProductService' with methods for creating products, getting product cache keys, and converting products to dictionaries for caching.

```
src > services > product_service.py
1 import logging
2 from typing import Any, Dict
3
4 from src.repositories.product_repository import ProductRepository
5 from src.schemas.product import ProductCreate, ProductUpdate
6 from src.services.cache_service import CacheService
7
8 logger = logging.getLogger(__name__)
9
10
11 class ProductService:
12     PRODUCT_CACHE_TTL = 600 # Cache TTL: 10 min
13
14     def __init__(self, product_repository: ProductRepository, cache_service: CacheService):
15         self.product_repository = product_repository
16         self.cache_service = cache_service
17
18     def _get_product_cache_key(self, product_id: int) -> str:
19         """Generate cache key for product"""
20         return f"product:{product_id}"
21
22     def _product_to_dict(self, product) -> dict:
23         """Convert Product model to dict for caching"""
24         return {
25             "id": product.id,
26             "name": product.name,
27             "price": float(product.price),
28             "stock_quantity": product.stock_quantity,
29         }
30
31     async def create(self, data: ProductCreate | dict) -> Dict[str, Any]:
32         payload = data.model_dump() if hasattr(data, "model_dump") else dict(data)
33         return await self.product_repository.create(**payload)
34
35     async def get_by_id(self, product_id: int):
36         """Get product by id (with caching)"""
37         cache_key = self._get_product_cache_key(product_id)
```

```

product_service.py
src > services > product_service.py
11     class ProductService:
35         async def get_by_id(self, product_id: int):
36             cached_data = await self.cache_service.get(cache_key)
37             if cached_data:
38                 logger.info(f"Cache HIT for product {product_id}")
39                 return cached_data
40
41             logger.info(f"Cache MISS for product {product_id}")
42             product = await self.product_repository.get_by_id(product_id)
43
44             if product:
45                 product_dict = self._product_to_dict(product)
46                 await self.cache_service.set(cache_key, product_dict, self.PRODUCT_CACHE_TTL)
47                 return product
48
49             return None
50
51
52
53
54         async def get_by_filter(self, count: int = 10, page: int = 1) -> Dict[str, Any]:
55             """Get product with page filter"""
56             return await self.product_repository.get_by_filter(count, page)
57
58         async def update(self, product_id: int, data: ProductUpdate | dict) -> Dict[str, Any]:
59             """Update product with caching"""
60             patch = (
61                 data.model_dump(exclude_none=True)
62                 if hasattr(data, "model_dump")
63                 else dict(data)
64             )
65
66             updated_product = await self.product_repository.update(product_id, **patch)
67
68             if updated_product:
69                 cache_key = self._get_product_cache_key(product_id)
70                 product_dict = self._product_to_dict(updated_product)
71                 await self.cache_service.set(cache_key, product_dict, self.PRODUCT_CACHE_TTL)
72                 logger.info(f"Cache UPDATED for product {product_id}")
73
74             return updated_product
75
76         async def delete(self, product_id: int) -> None:
77             """Delete product from db and cache"""
78             await self.product_repository.delete(product_id)
79
80             cache_key = self._get_product_cache_key(product_id)
81             await self.cache_service.delete(cache_key)
82             logger.info(f"Cache DELETED for product {product_id}")
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

Добавлено кэширование с требуемым временем жизни:

- PRODUCT_CACHE_TTL = 600 (10 минут = 600 секунд)
- т.к. данные о продуктах (особенно stock_quantity) меняются чаще, чем данные пользователей, поэтому требуется более короткий TTL для актуальности.

Полный цикл кэширования:

- Чтение (get_by_id) - проверка кэша, при промахе загрузка из БД и кэширование
- Обновление (update) - изменение в БД и обновление кэша

- Удаление (delete) - удаление из БД и удаление из кэша

Паттерны кэширования, которые используются в работе:

- Cache-Aside (Lazy Loading) - приложение проверяет кэш, при промахе загружает из БД
- Write-Through - при обновлении данные записываются и в БД, и в кэш
- Cache Invalidation - при удалении данные удаляются из кэша
- TTL-based Expiration - автоматическое истечение кэша для актуальности данных

Ответы на вопросы

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

Главное преимущество — скорость. Оперативная память даёт значительно меньшие задержки и более высокую пропускную способность для чтения/записи по сравнению с дисковыми носителями. Это делает in-memory БД (например, Redis), подходящими для кэширования, очередей, сессий и операций с высокими требованиями к латентности.

Дополнительные преимущества:

- Низкая задержка (latency) - подходит для real-time приложений
- Высокая пропускная способность - Redis обрабатывает 100 000+ операций/сек на обычном сервере, а PostgreSQL: ~10 000 операций/сек
- Предсказуемая производительность - нет задержек в поиске по индексам

Ограничение - стоимость и ограниченность объёма RAM (обычный сервер вмещает 16-128 GB RAM), а также RAM энергозависима и для долговременного хранения обычно добавляют AOF/RDB персистентность.

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

Поскольку Redis хранит всё как бинарные данные (bytes), то по умолчанию ответы от сервера возвращаются в виде байтов. Параметр decode_responses=True говорит клиенту автоматически декодировать байты в строки (str) используя указанную кодировку (по умолчанию UTF-8). Это удобно при работе в Python т.к. не нужно вручную декодировать ключи/значения.

Я также использую decode_responses=False в этой работе, т.к. Redis используется как байтовое хранилище, а кодировка и формат данных полностью контролируются приложением, а не клиентом Redis. Redis всегда хранит данные как bytes, а мы далее сериализуем данные в JSON, поэтому важно получать их в виде bytes, без автоматических преобразований.

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

TTL - время жизни ключа, обычно задается в секундах или миллисекундах. Своего рода таймер автоматического удаления , если у ключа установлен TTL, по истечении этого времени ключ автоматически удаляется сервером. Команды: EXPIRE/PEXPIRE (установить), TTL/PTTL (узнать оставшееся время), PERSIST (удалить таймаут — сделать вечным). TTL используют для кэширования, временных сессий, счетчиков и т.п. TTL позволяет Redis самостоятельно очищать устаревшие данные без участия приложения, снижая потребление памяти и упрощая логику системы.

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

- LPUSH key value [value ...] — добавляет элементы в начало (левый конец) списка. При передаче нескольких значений они добавляются последовательно, поэтому порядок в списке будет обратным порядку аргументов (например, LPUSH mylist a b c → список c, b, a).
- RPUSH key value [value ...] — добавляет элементы в конец (правый конец) списка; при нескольких аргументах порядок сохраняется (RPUSH mylist a b c → a, b, c).

Операции работают за константное время $O(1)$.

5. Как обеспечить атомарность операций в Redis?

Атомарная операция - операция, которая выполняется полностью или не выполняется вообще, без промежуточных состояний.

Способы обеспечения атомарности операций:

- 1) **Атомарные команды Redis** предоставляет собой встроенные атомарные операции - счетчики: INCR, DECR, INCRBY, DECRBY. самый простой и быстрый путь, если вся логика укладывается в одну команду.
- 2) **Транзакции (MULTI/EXEC)** представляют собой группировку нескольких команд в одну атомарную операцию. MULTI начинает транзакцию, команды ставятся в очередь, EXEC выполняет все queued-команды последовательно как непрерывную операцию. Транзакция гарантирует, что все команды из очереди выполняются подряд без интервенций других клиентов между ними, но не обеспечивает отката в случае ошибки внутри команды.
- 3) **Lua-скрипты** - выполняются целиком атомарно на стороне Redis.
Преимущества: весь скрипт выполняется атомарно, никакие другие клиенты не могут вмешаться, можно использовать сложную логику (н-р циклы), минимум сетевых запросов (т.к. скрипт на стороне сервера)
- 4) **Optimistic Locking (WATCH)** - WATCH отмечает ключ для отслеживания, если ключ изменится до EXEC, то транзакция отменится. Приложение может повторить попытку
- 5) **Распределённые блокировки (н-р Redlock)** - используются, если нужна блокировка среди нескольких клиентов/процессов/машин.

6. Как в Redis реализована репликация и кластеризация?

Репликация - создание копий данных на нескольких серверах для

повышения доступности и распределения чтения.

Как это работает:

- Master принимает все операции записи (SET, INCR, DEL и т.д.) и транслирует (реплицирует) эти изменения в поток команд для реплик.
- Replica подключается к мастеру, получает поток команд и выполняет их локально - таким образом данные синхронизируются. Реплики обычно используются для операций чтения (GET, LRANGE и т.п.).
- Реплики по умолчанию не принимают записи от клиентов (read-only).

Записи направляются на мастера.

Репликация в Redis асинхронная: мастер отправляет обновления, но запись считается завершённой с точки зрения клиента сразу после записи на мастере (реплики могут отставать). Реплики увеличивают пропускную способность чтения и повышают отказоустойчивость (при выходе мастера одна из реплик может быть промодирована в мастер). Для автоматического мониторинга и failover используется отдельный компонент (например, Redis Sentinel), который наблюдает за узлами и при необходимости выполняет переключение ролей.

Кластеризация - механизм горизонтального масштабирования Redis, при котором данные распределяются между несколькими узлами кластера.

- В Redis Cluster данные делятся на 16384 hash-слота.
- Каждый ключ попадает в определённый слот на основе хэша ключа.
- Каждый узел кластера отвечает за набор слотов.

Как работает кластер:

- Клиент отправляет команду любому узлу кластера.
- Если узел не обслуживает нужный слот, он возвращает клиенту перенаправление (MOVED).
- Клиент повторяет запрос на правильный узел.
- Каждый primary-узел может иметь одну или несколько реплик

Redis Cluster поддерживает горизонтальное масштабирование (данные и нагрузка распределяются) и обеспечивает автоматический failover: если primary-узел падает, его реплика становится новым primary. Запись и чтение масштабируются по нескольким узлам.

Вывод:

В ходе выполнения лабораторной работы были успешно освоены основы работы с Redis. Была добавлена и настроена инфраструктура Redis в виде контейнера Docker, что позволило легко развернуть in-memory базу данных. Отработаны навыки подключения к Redis из Python-приложения и проверки соединения. Изучены основные структуры данных Redis: строки, списки, множества, хэши и упорядоченные множества. Практически применены команды для работы с этими структурами, включая операции установки, получения, модификации и удаления данных. Особое внимание уделено работе с TTL (временем жизни ключей), что важно для реализации кэширования.

В существующее приложение было интегрировано кэширование данных о пользователях и продукции с разным временем жизни. Реализована инвалидация кэша при обновлении данных, что обеспечивает актуальность информации. Это позволило значительно снизить нагрузку на основную базу данных и ускорить обработку запросов.

Таким образом, лабораторная работа позволила получить практические навыки работы с Redis, понять его преимущества как in-memory хранилища и освоить основные приемы использования Redis для кэширования в веб-приложениях.