

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.

Екатеринбург 2025

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 X

docker-compose.yml

Run All Services

services:

Run Service

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

Run Service

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 X

docker-compose.yml

services:

db:

healthcheck:

Run Service

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

Run Service

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

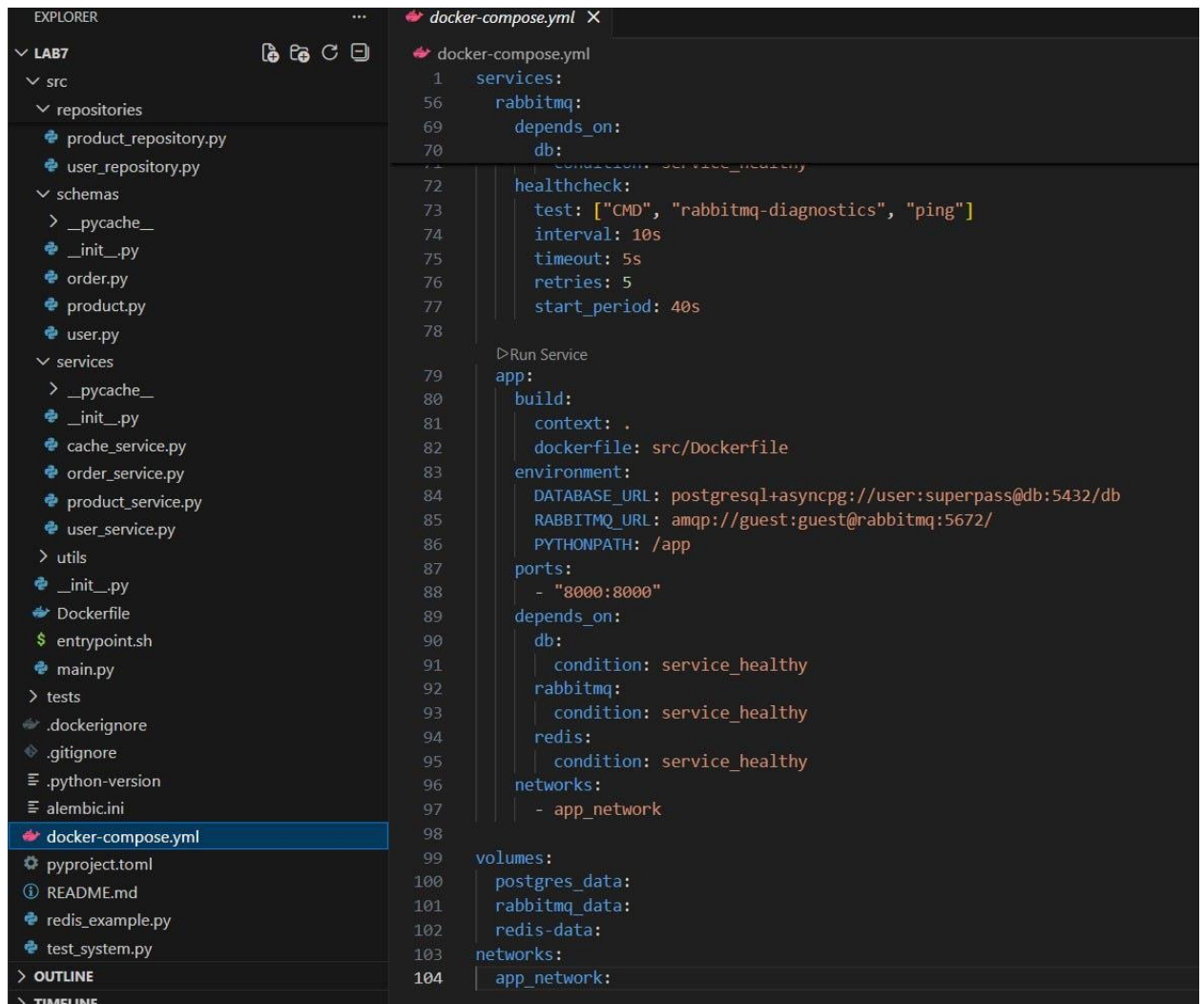
networks:

- app_network

depends_on:

db:

condition: service_healthy



Ход выполнения:

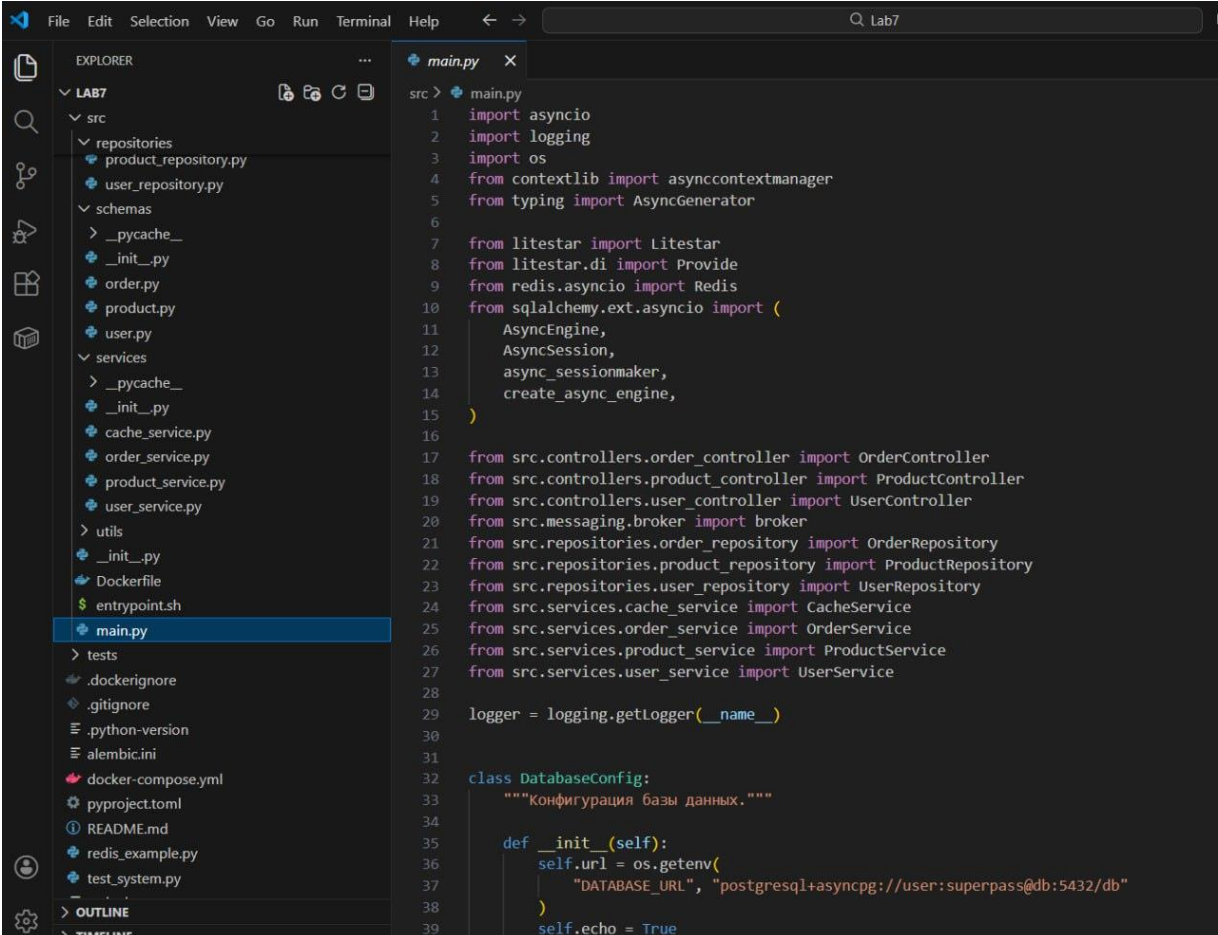
Был добавлен сервис Redis с полной конфигурацией:

- 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*

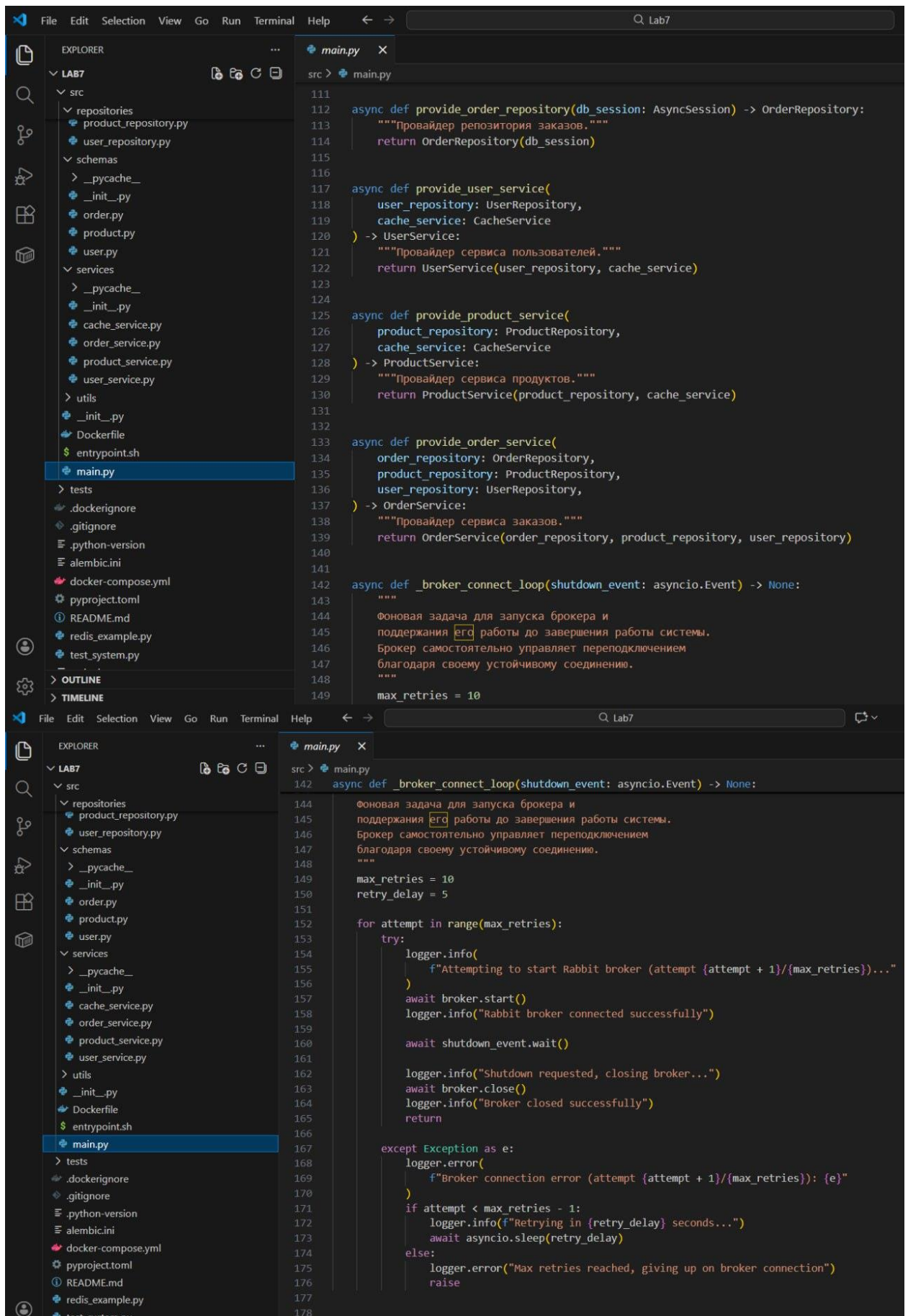


The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays a project structure for 'LAB7'. The 'src' directory contains subdirectories 'repositories', 'schemas', and 'services', each with an '__init__.py' file and specific service/controller files. The 'main.py' file is selected and open in the editor. The code in 'main.py' includes imports for 'asyncio', 'logging', 'os', 'contextlib', 'typing', 'litestar', 'redis', and 'sqlalchemy'. It defines a 'DatabaseConfig' class with an '__init__' method that sets the database URL from an environment variable. The code also shows imports for various controllers and services from the 'src' subdirectories.

```
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
```

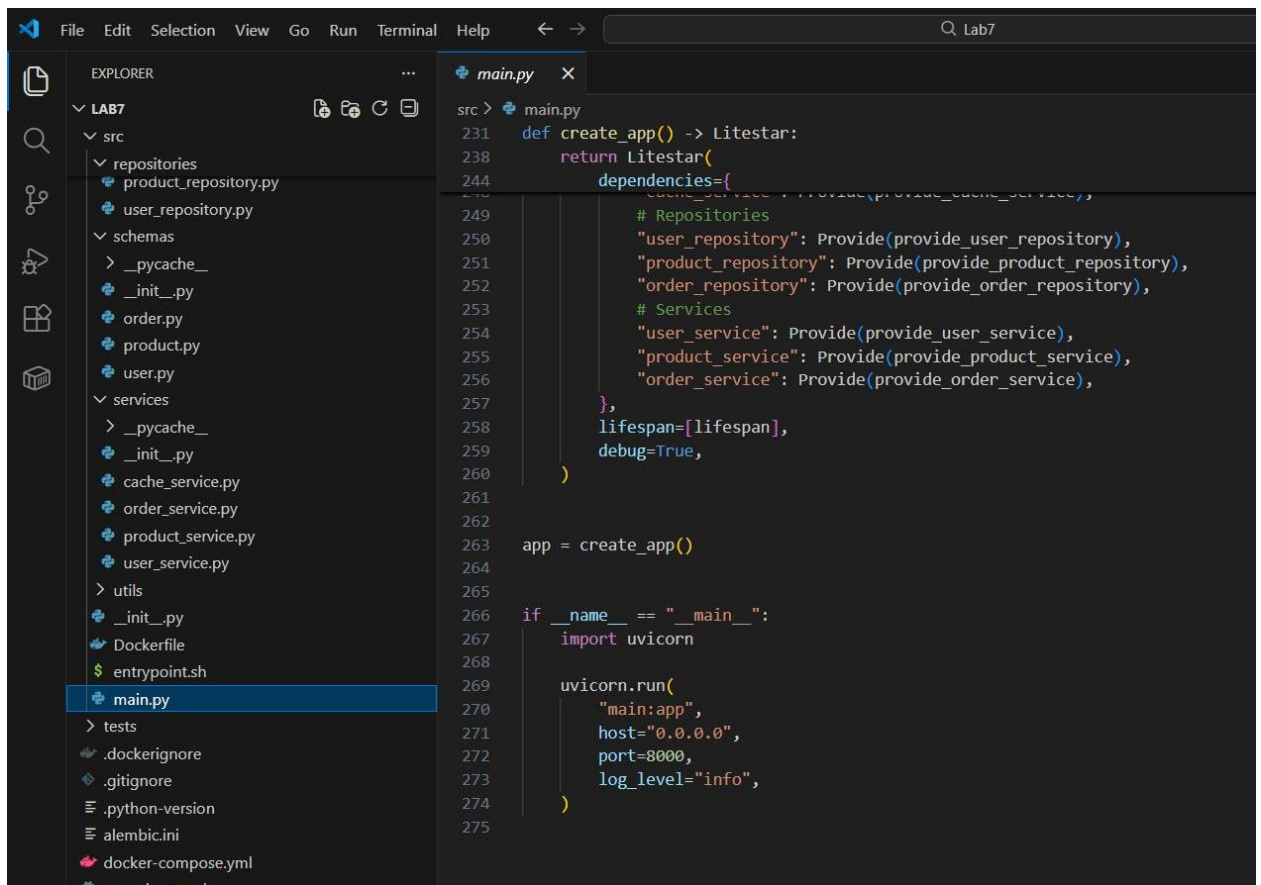
```
32 class DatabaseConfig:
33     def __init__(self):
34
35
36
37
38
39
40
41     def create_engine(self) -> AsyncEngine:
42         """Создает engine для подключения к БД."""
43         return create_async_engine(
44             self.url,
45             echo=self.echo,
46             pool_pre_ping=True,
47             pool_size=10,
48             max_overflow=20,
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
61     def create_client(self) -> Redis:
62         """Создает клиент Redis."""
63         return Redis(
64             host=self.host,
65             port=self.port,
66             db=self.db,
67             decode_responses=False,
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,
```

```
72 async_session_maker = async_sessionmaker(
73     engine,
74     class_=AsyncSession,
75     expire_on_commit=False,
76     autoflush=False,
77 )
78
79 redis_config = RedisConfig()
80 redis_client: Redis = None
81
82
83 async def provide_db_session() -> AsyncGenerator[AsyncSession, None]:
84     """
85     Провайдер сессии базы данных.
86
87     Yields:
88         AsyncSession: Активная сессия БД
89     """
90     async with async_session_maker() as session:
91         try:
92             yield session
93         finally:
94             await session.close()
95
96
97 async def provide_cache_service() -> CacheService:
98     """Провайдер сервиса кеширования."""
99     return CacheService(redis_client)
100
101
102 async def provide_user_repository(db_session: AsyncSession) -> UserRepository:
103     """Провайдер репозитория пользователей."""
104     return UserRepository(db_session)
105
106
107 async def provide_product_repository(db_session: AsyncSession) -> ProductRepository:
108     """Провайдер репозитория продуктов."""
109     return ProductRepository(db_session)
110
111
112 async def provide_order_repository(db_session: AsyncSession) -> OrderRepository:
113     """Провайдер репозитория заказов."""
114     return OrderRepository(db_session)
```

```
178
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:
```

```
217         except asyncio.CancelledError:
218             logger.exception("Broker task raised while shutting down")
219
220         try:
221             await redis_client.close()
222             logger.info("Redis connection closed")
223         except Exception:
224             logger.exception("Error closing Redis connection")
225
226         try:
227             await engine.dispose()
228         except Exception:
229             logger.exception("Error disposing engine")
230
231     def create_app() -> Litestar:
232         """
233         Фабрика для создания приложения Litestar.
234
235         Returns:
236             Litestar: Сконфигурированное приложение
237         """
238         return Litestar(
239             route_handlers=[
240                 UserController,
241                 ProductController,
242                 OrderController,
243             ],
244             dependencies={
245                 # Database
246                 "db_session": Provide(provide_db_session),
247                 # Cache
248                 "cache_service": Provide(provide_cache_service),
249                 # Repositories
250                 "user_repository": Provide(provide_user_repository),
251                 "product_repository": Provide(provide_product_repository),
```

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:`

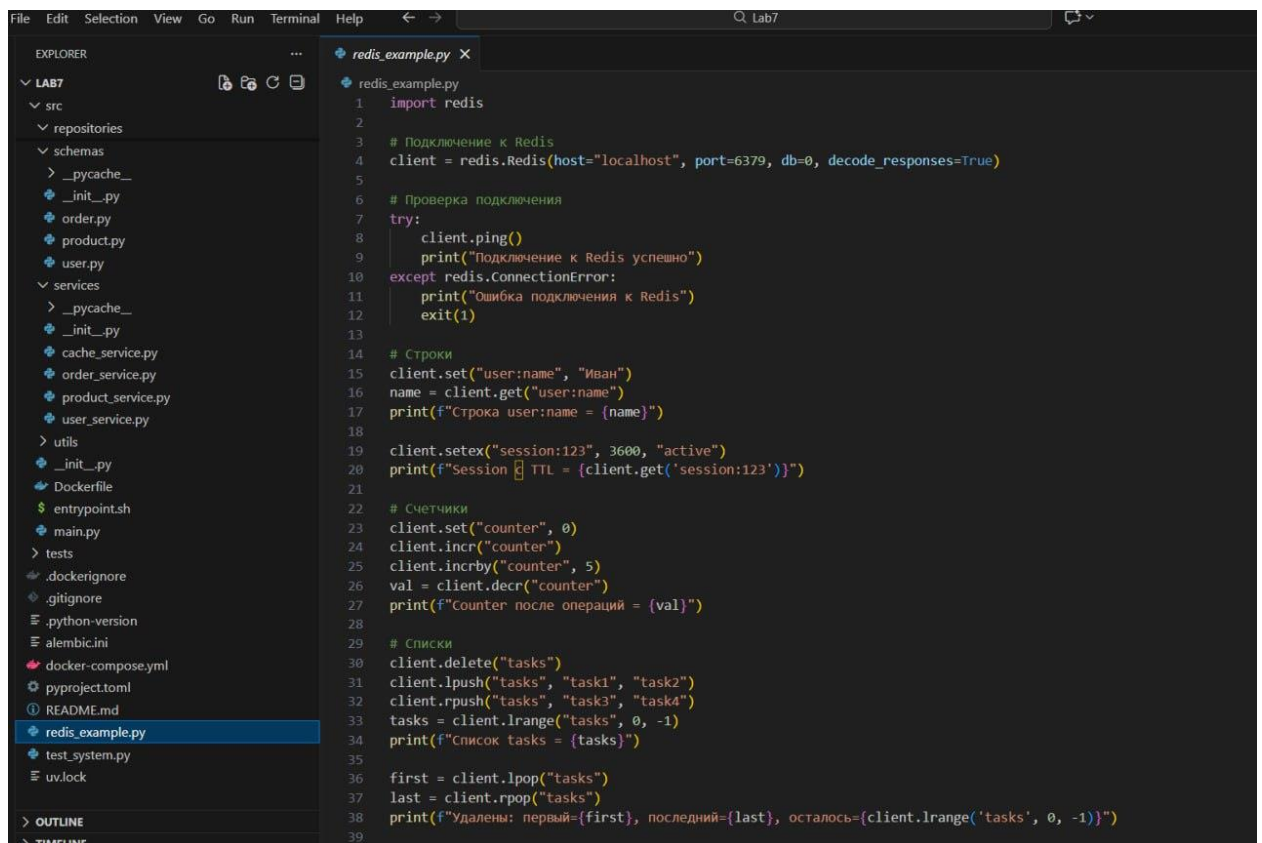
```
"""Провайдер сервиса кэширования."""
return CacheService(redis_client)
```

- 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)



```
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)}")
39
```

```
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(f"Все операции выполнены успешно")
67
```

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

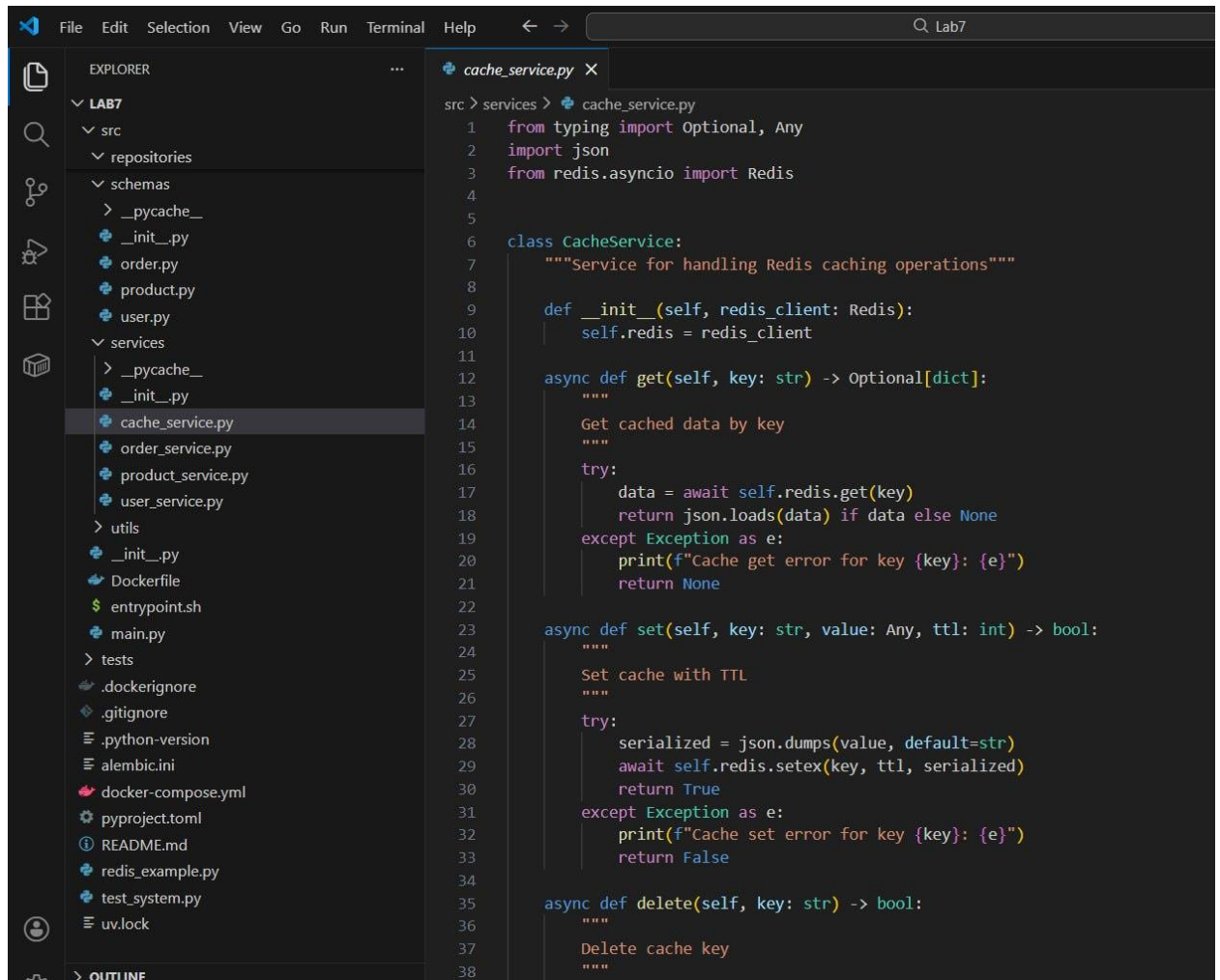
Пример запуска теста:

```
(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>
```

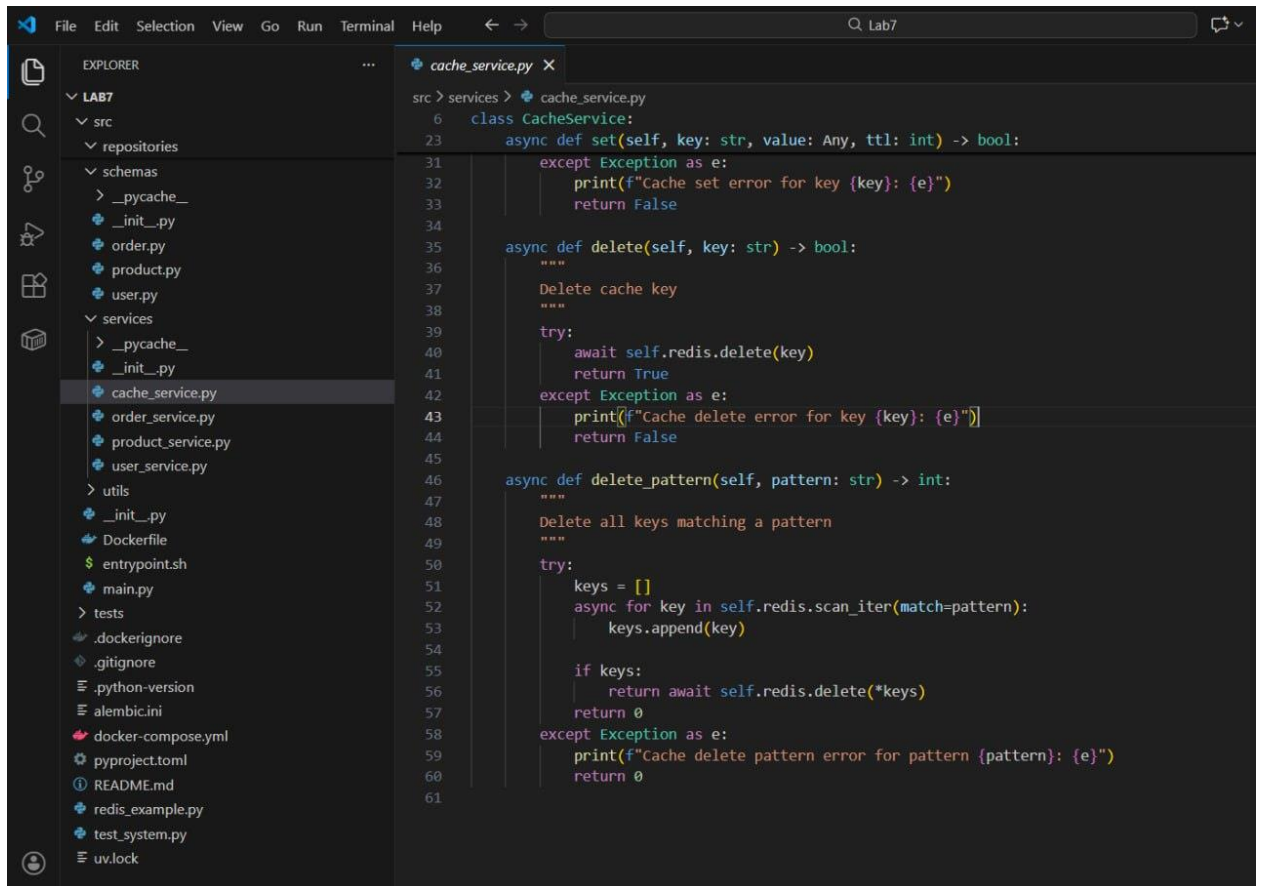
Часть 4: Кэширование запросов и очищение запросов

Файл: *cache_service.py*

(Создание сервиса кэширования)



```
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         """
```



```
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         try:
40             await self.redis.delete(key)
41             return True
42         except Exception as e:
43             print(f"Cache delete error for key {key}: {e}")
44             return False
45
46     async def delete_pattern(self, pattern: str) -> int:
47         """
48         Delete all keys matching a pattern
49         """
50         try:
51             keys = []
52             async for key in self.redis.scan_iter(match=pattern):
53                 keys.append(key)
54
55             if keys:
56                 return await self.redis.delete(*keys)
57             return 0
58         except Exception as e:
59             print(f"Cache delete pattern error for pattern {pattern}: {e}")
60             return 0
61
```

Ход выполнения:

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

Основные методы:

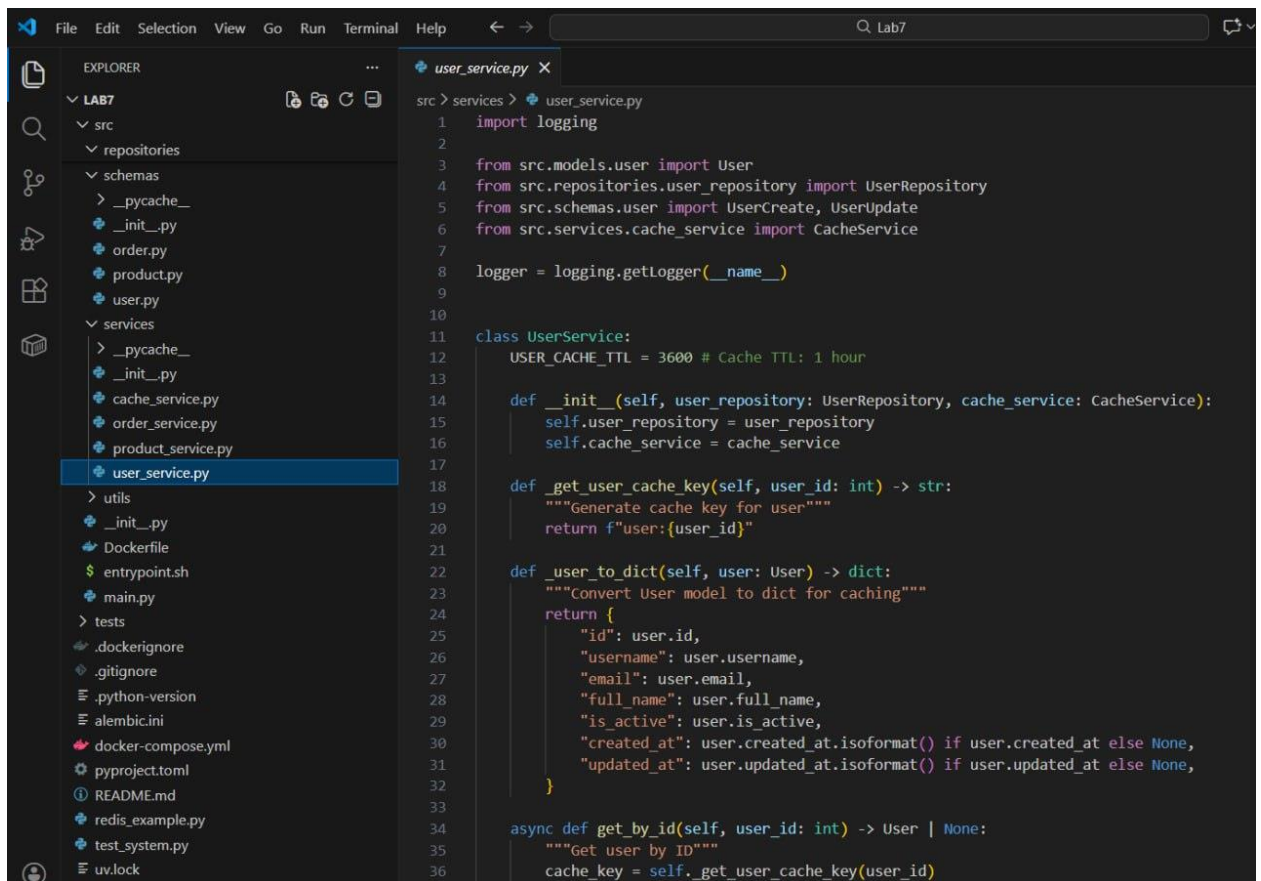
- **get(key)** - получение данных с десериализацией из JSON. Метод Получает bytes из Redis и десериализует JSON в Python dict. При ошибке возвращает None (приложение продолжит работу без кэша)
- **set(key, value, ttl)** - сохранение в кэш с автоматическим истечением.
 - `json.dumps(value, default=str)` - 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-ехсепт 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))



```
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
11 class UserService:
12     USER_CACHE_TTL = 3600 # Cache TTL: 1 hour
13
14     def __init__(self, user_repository: UserRepository, cache_service: CacheService):
15         self.user_repository = user_repository
16         self.cache_service = cache_service
17
18     def _get_user_cache_key(self, user_id: int) -> str:
19         """Generate cache key for user"""
20         return f"user:{user_id}"
21
22     def _user_to_dict(self, user: User) -> dict:
23         """Convert User model to dict for caching"""
24         return {
25             "id": user.id,
26             "username": user.username,
27             "email": user.email,
28             "full_name": user.full_name,
29             "is_active": user.is_active,
30             "created_at": user.created_at.isoformat() if user.created_at else None,
31             "updated_at": user.updated_at.isoformat() if user.updated_at else None,
32         }
33
34     async def get_by_id(self, user_id: int) -> User | None:
35         """Get user by ID"""
36         cache_key = self._get_user_cache_key(user_id)
```

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

Добавлено кэширование с требуемым временем жизни:

- `USER_CACHE_TTL = 3600` (1 час = 3600 секунд)

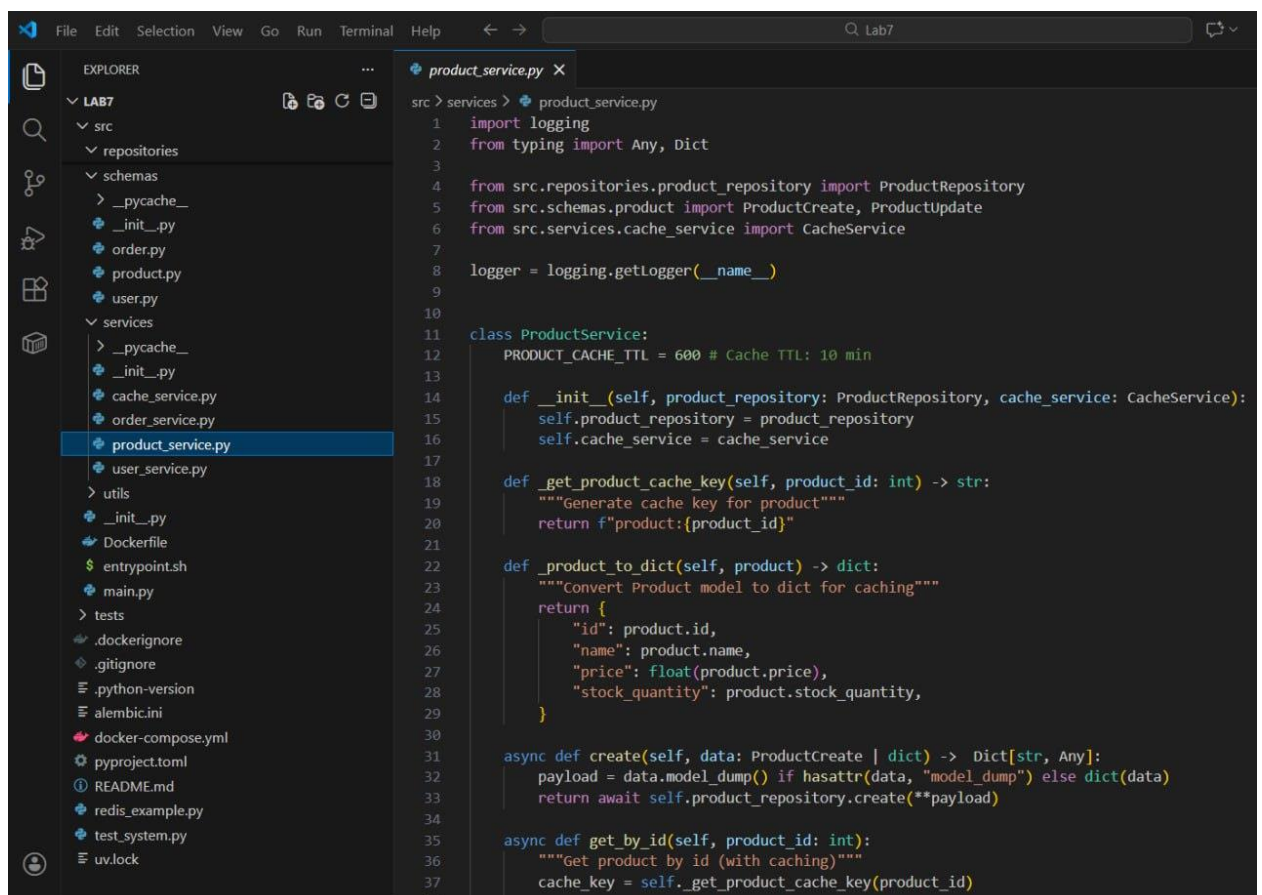
Вспомогательные методы:

- `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))



```

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)
38

```

```
src > services > product_service.py
11 class ProductService:
35     async def get_by_id(self, product_id: int):
39         cached_data = await self.cache_service.get(cache_key)
40         if cached_data:
41             logger.info(f"Cache HIT for product {product_id}")
42             return cached_data
43
44         logger.info(f"Cache MISS for product {product_id}")
45         product = await self.product_repository.get_by_id(product_id)
46
47         if product:
48             product_dict = self._product_to_dict(product)
49             await self.cache_service.set(cache_key, product_dict, self.PRODUCT_CACHE_TTL)
50             return product
51
52         return None
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
```

```
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
```

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).
- `R PUSH key value [value ...]` — добавляет элементы в конец (правый конец) списка; при нескольких аргументах порядок сохраняется (`R PUSH 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 выполнятся 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 промотирована 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.