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

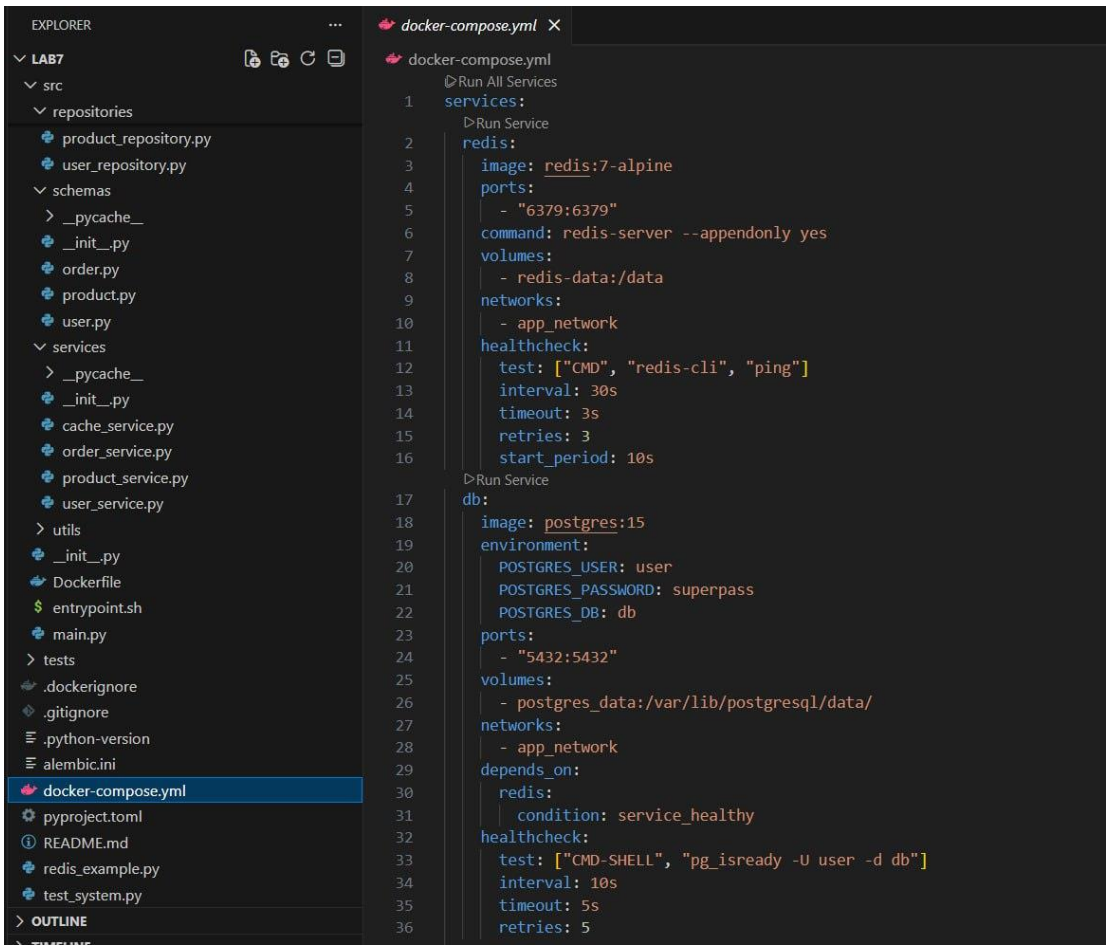
Овладеть базовыми навыками установки, подключения и взаимодействия с Redis в Python. Изучить основные структуры данных Redis и их применение на практике.

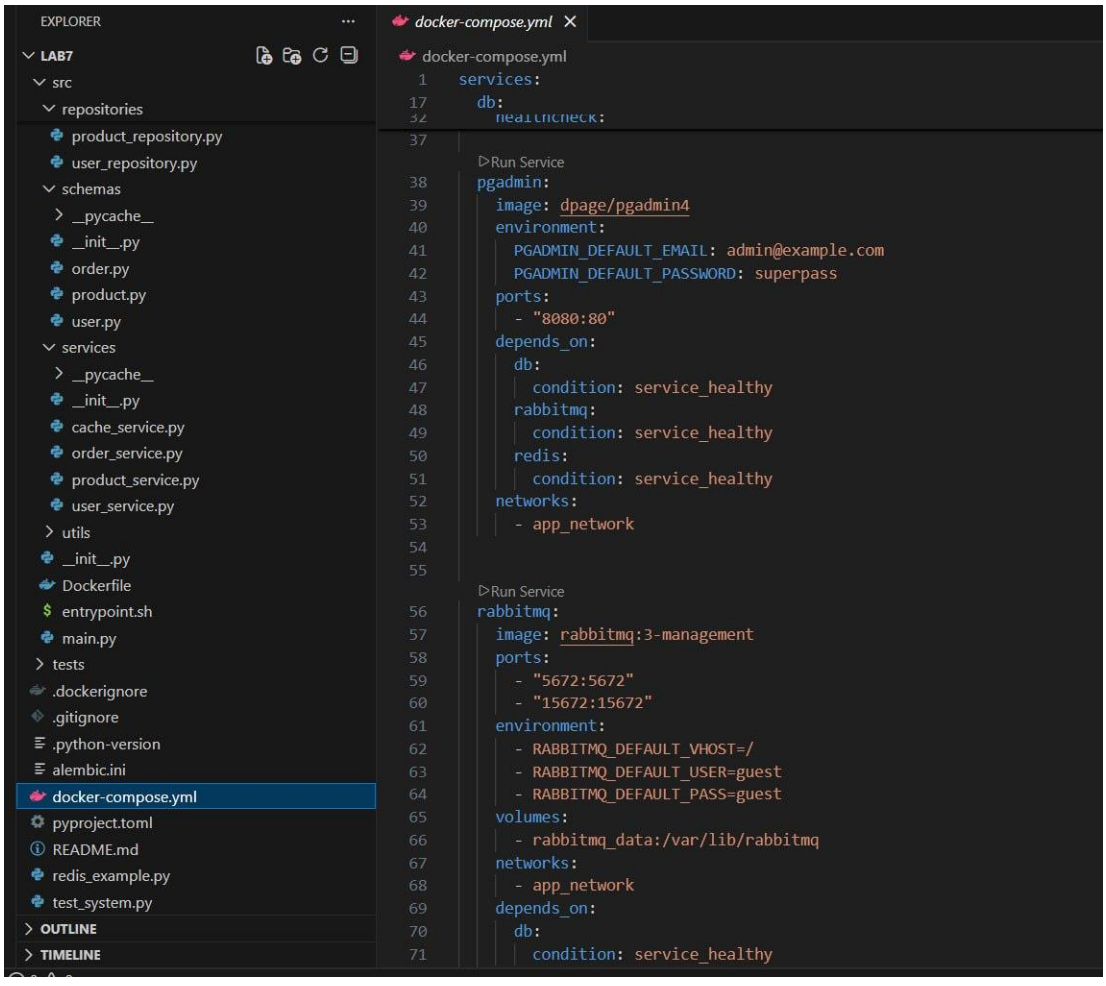
Задачи

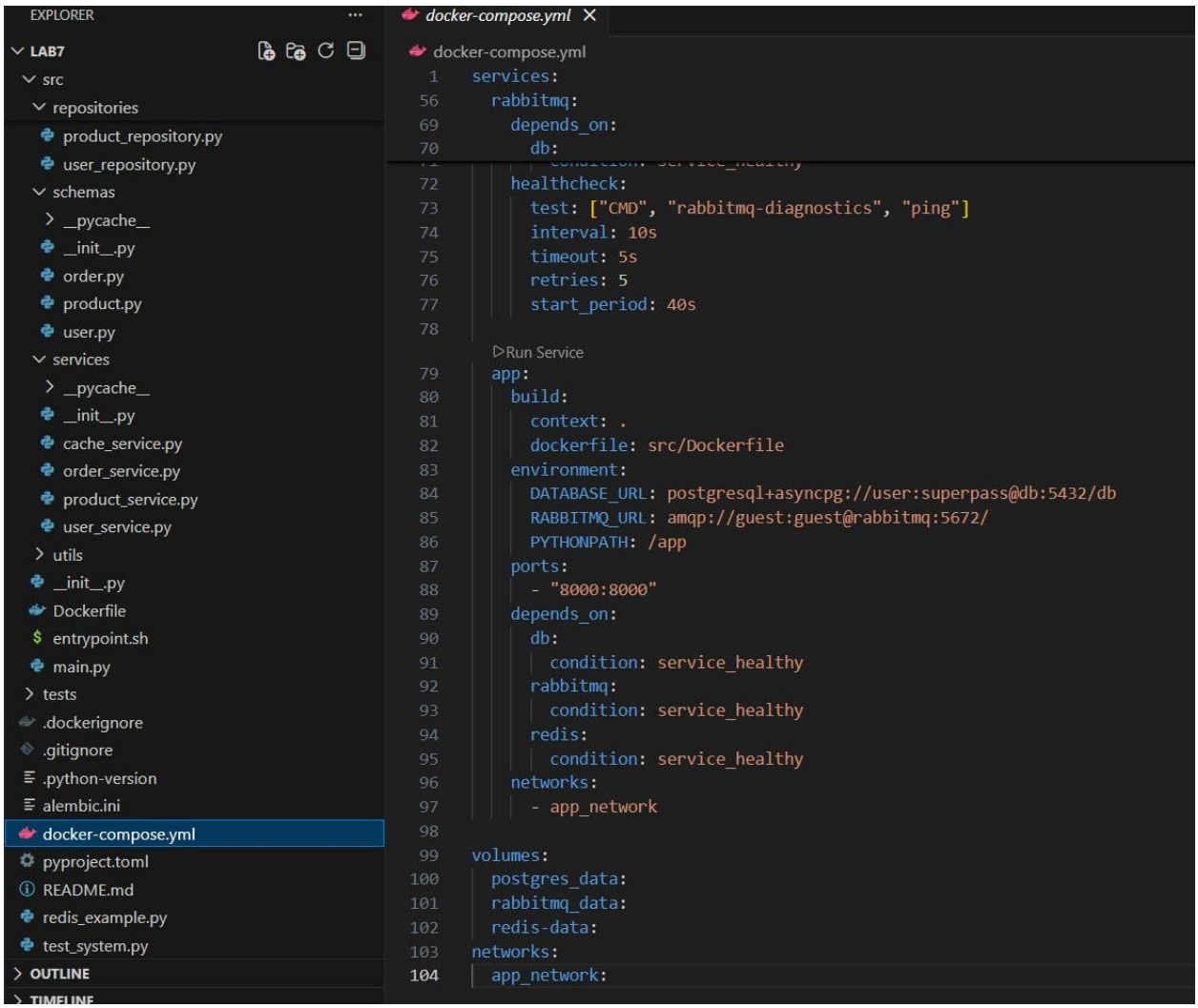
1. Добавить контейнер Redis в docker-compose и настроить подключение к Redis из Python-приложения.
2. Изучить работу с основными структурами данных Redis: строки (установка, получение, TTL), списки (добавление, получение элементов), множества, хэши, упорядоченные множества.
3. Реализовать кэширование 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
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
services:
db:
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:

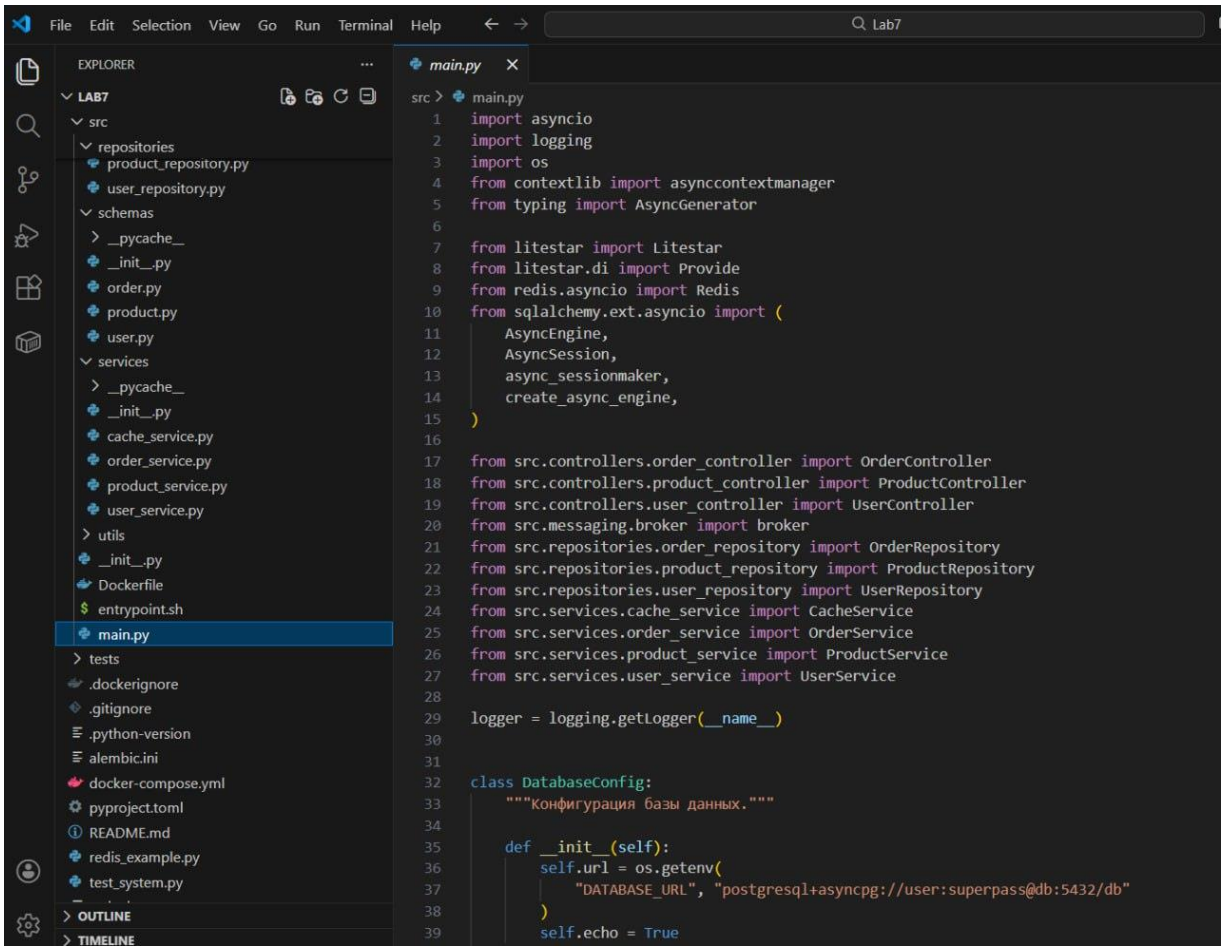
Был добавлен сервис Redis с полной конфигурацией:

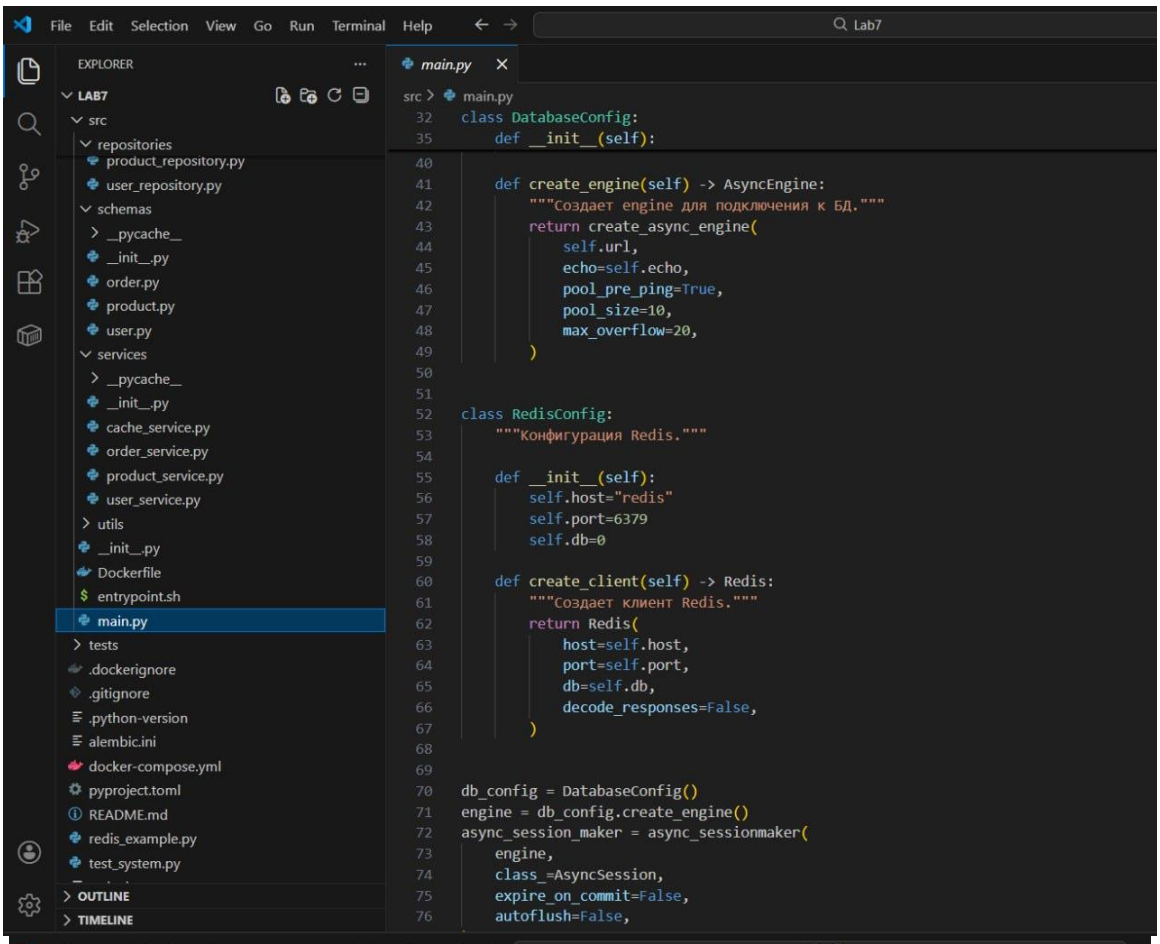
- image: redis:7-alpine - использовали облегченную версию Redis 7
- ports: "6379:6379" - стандартный порт Redis
- command: redis-server --appendonly yes - включили персистентность данных, чтобы данные сохранялись на диск (по умолчанию Redis хранит данные только в памяти)
- volumes: redis-data:/data - 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

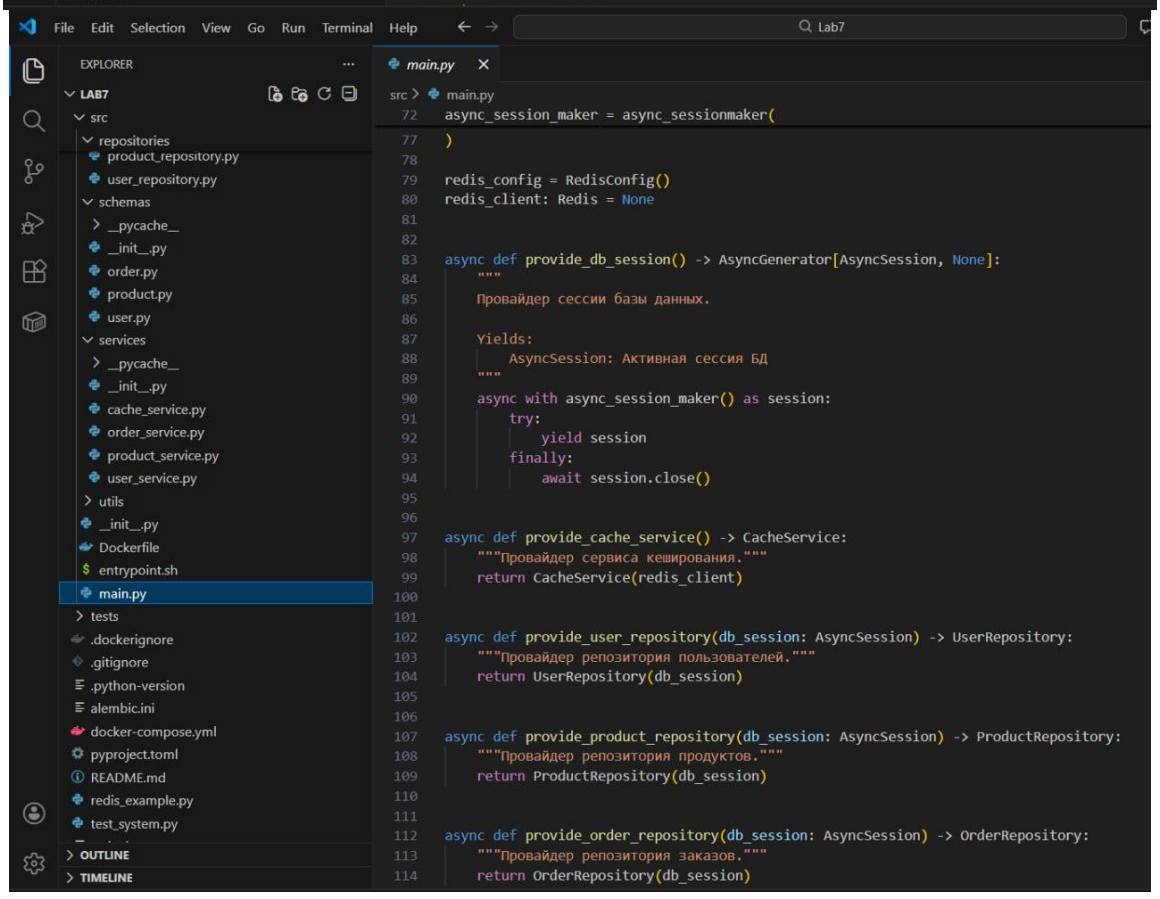
- networks: app_network - подключение к общей сети приложения для взаимодействия с другими сервисами

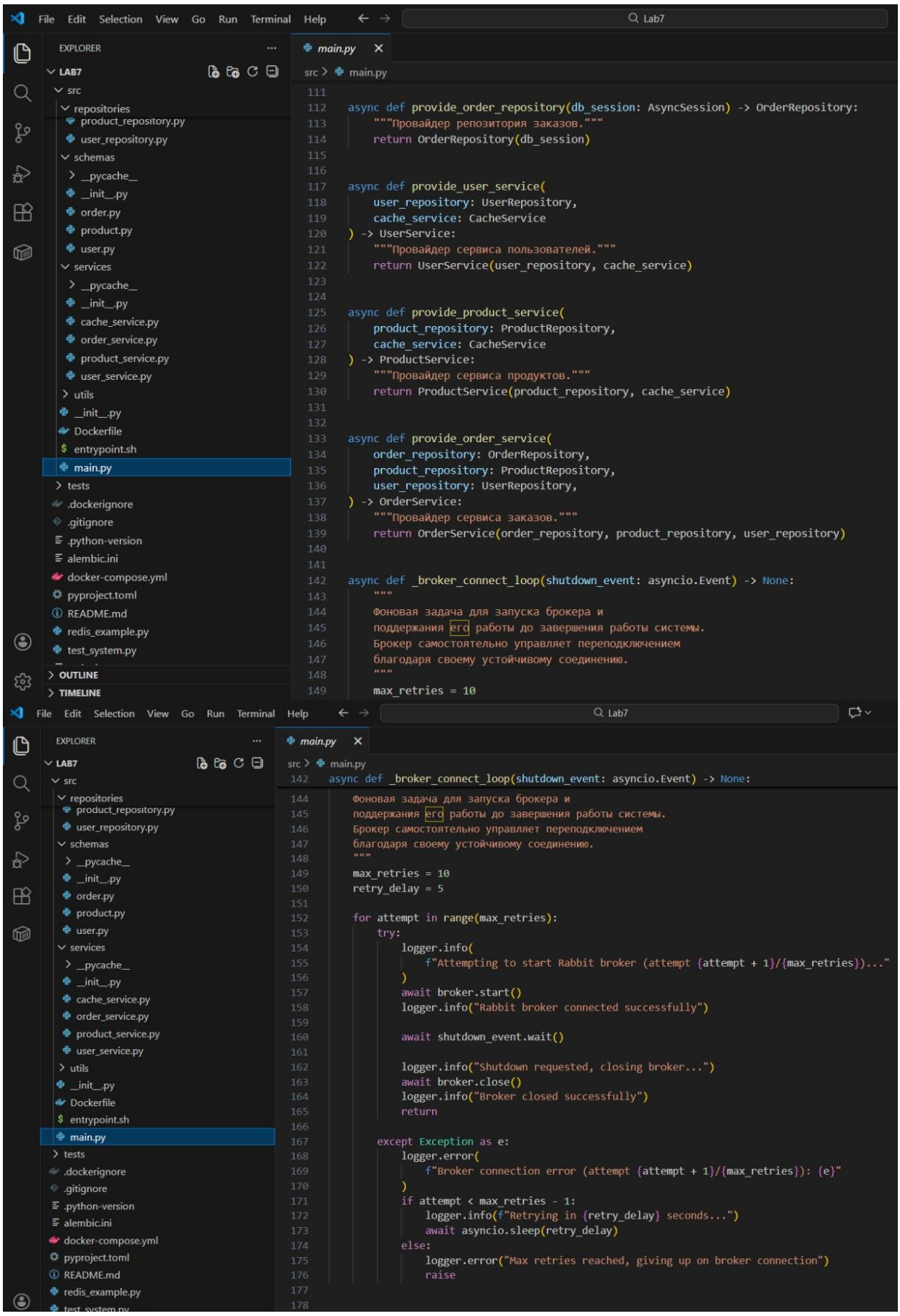
Часть 2: Подключение к Redis

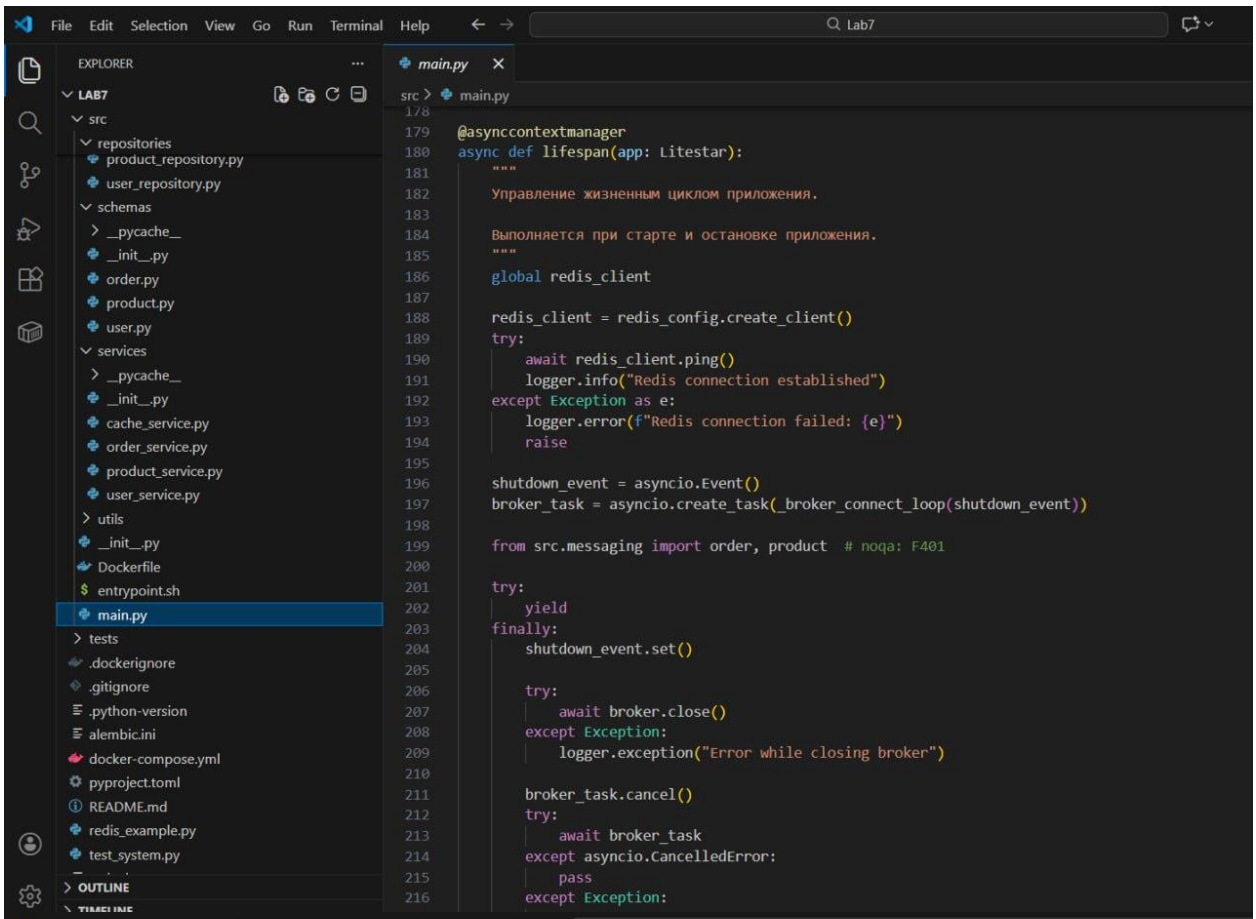
Файл: *main.py*

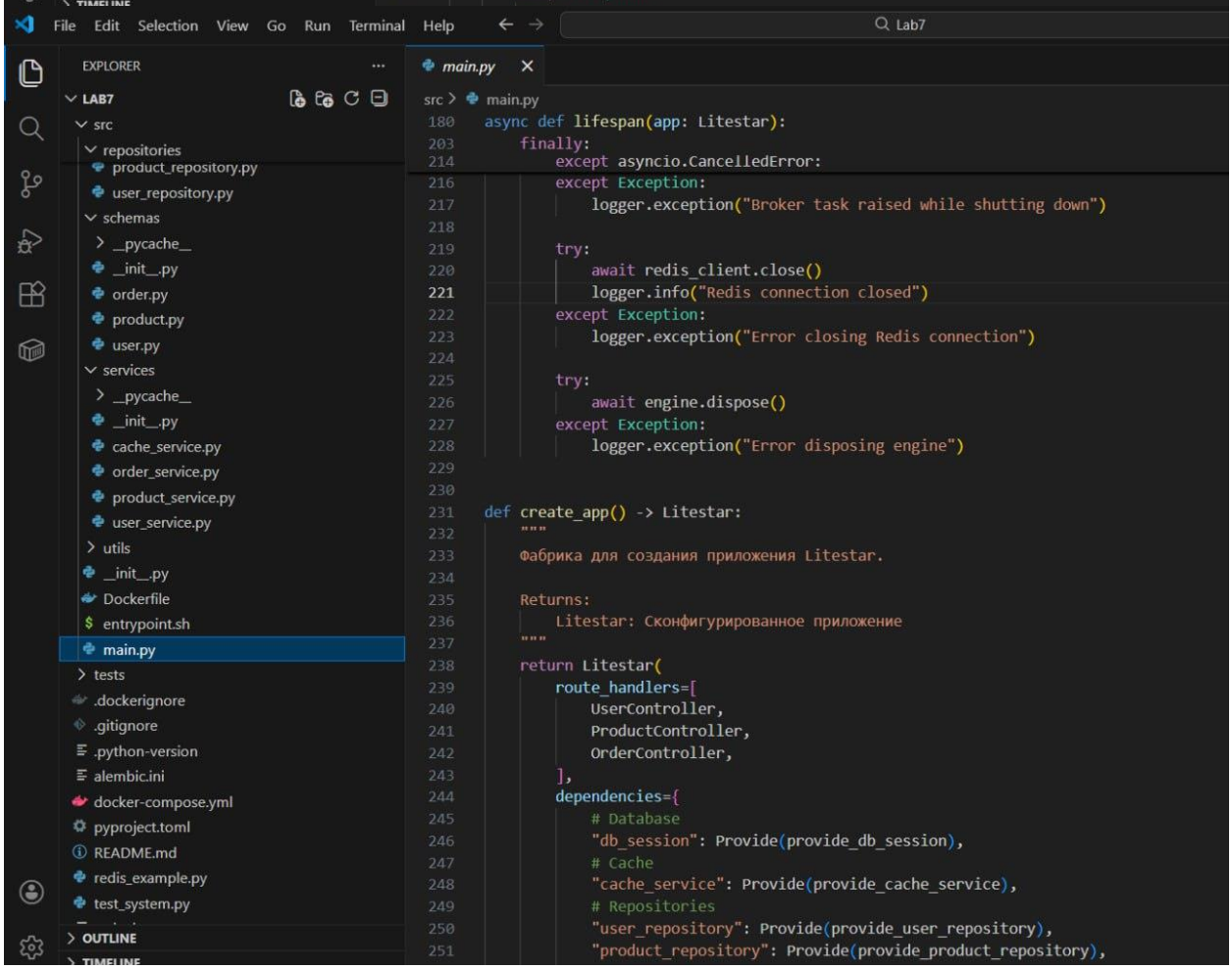


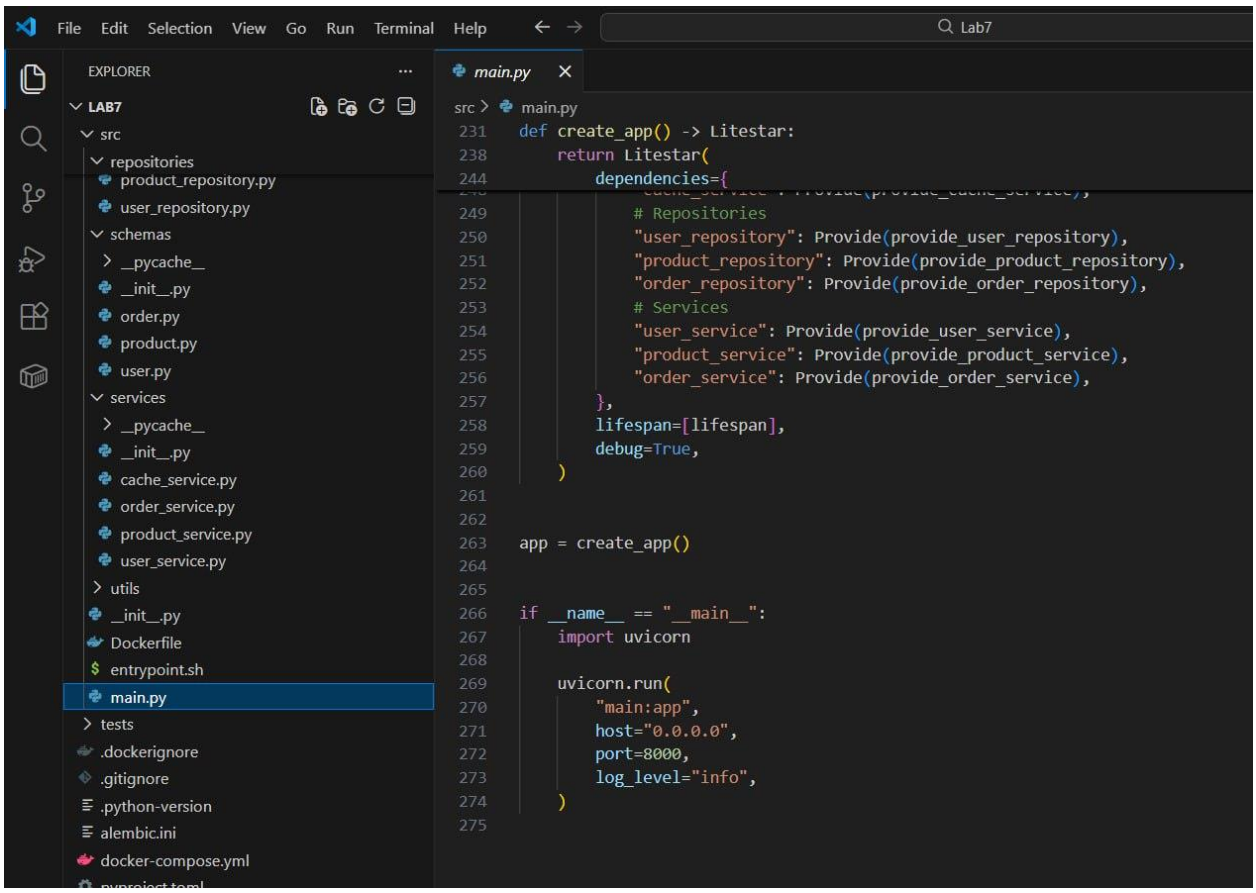

File Edit Selection View Go Run Terminal Help
Lab7
EXPLORER
LAB7
src
repositories
product_repository.py
user_repository.py
schemas
__pycache__
__init__.py
order.py
product.py
user.py
services
__pycache__
__init__.py
cache_service.py
order_service.py
product_service.py
user_service.py
utils
__init__.py
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
src > main.py
class DatabaseConfig:
def __init__(self):
def create_engine(self) -> AsyncEngine:
"""Создает engine для подключения к БД."""
return create_async_engine(
self.url,
echo=self.echo,
pool_pre_ping=True,
pool_size=10,
max_overflow=20,
)
class RedisConfig:
"""Конфигурация Redis."""
def __init__(self):
self.host="redis"
self.port=6379
self.db=0
def create_client(self) -> Redis:
"""Создает клиент Redis."""
return Redis(
host=self.host,
port=self.port,
db=self.db,
decode_responses=False,
)
db_config = DatabaseConfig()
engine = db_config.create_engine()
async_session_maker = async_sessionmaker(
engine,
class_=AsyncSession,
expire_on_commit=False,
autoflush=False,


File Edit Selection View Go Run Terminal Help
Lab7
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
src > main.py
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
Lab7
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
src > main.py
111
112 async def provide_order_repository(db_session: AsyncSession) -> OrderRepository:
113 """Провайдер репозитория заказов."""
114 return OrderRepository(db_session)
115
116
117 async def provide_user_service(
118 user_repository: UserRepository,
119 cache_service: CacheService
120) -> UserService:
121 """Провайдер сервиса пользователей."""
122 return UserService(user_repository, cache_service)
123
124
125 async def provide_product_service(
126 product_repository: ProductRepository,
127 cache_service: CacheService
128) -> ProductService:
129 """Провайдер сервиса продуктов."""
130 return ProductService(product_repository, cache_service)
131
132
133 async def provide_order_service(
134 order_repository: OrderRepository,
135 product_repository: ProductRepository,
136 user_repository: UserRepository,
137) -> OrderService:
138 """Провайдер сервиса заказов."""
139 return OrderService(order_repository, product_repository, user_repository)
140
141
142 async def _broker_connect_loop(shutdown_event: asyncio.Event) -> None:
143 """
144 Фоновая задача для запуска брокера и
145 поддержания его работы до завершения работы системы.
146 Брокер самостоятельно управляет переподключением
147 благодаря своему устойчивому соединению.
148 """
149 max_retries = 10
File Edit Selection View Go Run Terminal Help
Lab7
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
main.py
src > main.py
142 async def _broker_connect_loop(shutdown_event: asyncio.Event) -> None:
144 Фоновая задача для запуска брокера и
145 поддержания его работы до завершения работы системы.
146 Брокер самостоятельно управляет переподключением
147 благодаря своему устойчивому соединению.
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


File Edit Selection View Go Run Terminal Help
Lab7
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
src > main.py
@asynccontextmanager
async def lifespan(app: Litestar):
"""
Управление жизненным циклом приложения.
Выполняется при старте и остановке приложения.
"""
global redis_client
redis_client = redis_config.create_client()
try:
await redis_client.ping()
logger.info("Redis connection established")
except Exception as e:
logger.error(f"Redis connection failed: {e}")
raise
shutdown_event = asyncio.Event()
broker_task = asyncio.create_task(_broker_connect_loop(shutdown_event))
from src.messaging import order, product # noqa: F401
try:
yield
finally:
shutdown_event.set()
try:
await broker.close()
except Exception:
logger.exception("Error while closing broker")
broker_task.cancel()
try:
await broker_task
except asyncio.CancelledError:
pass
except Exception:


File Edit Selection View Go Run Terminal Help
Lab7
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
src > main.py
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),



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

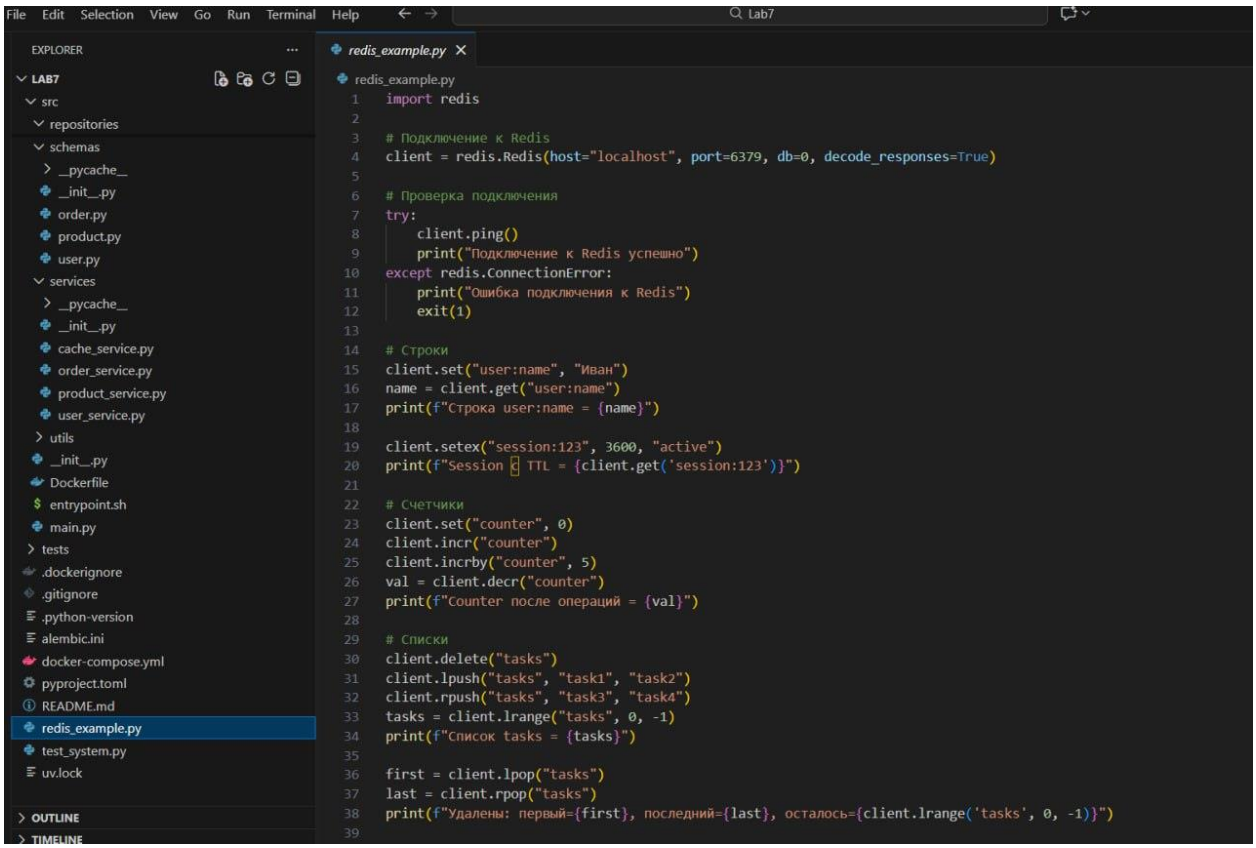
3. Провайдер для внедрения зависимостей

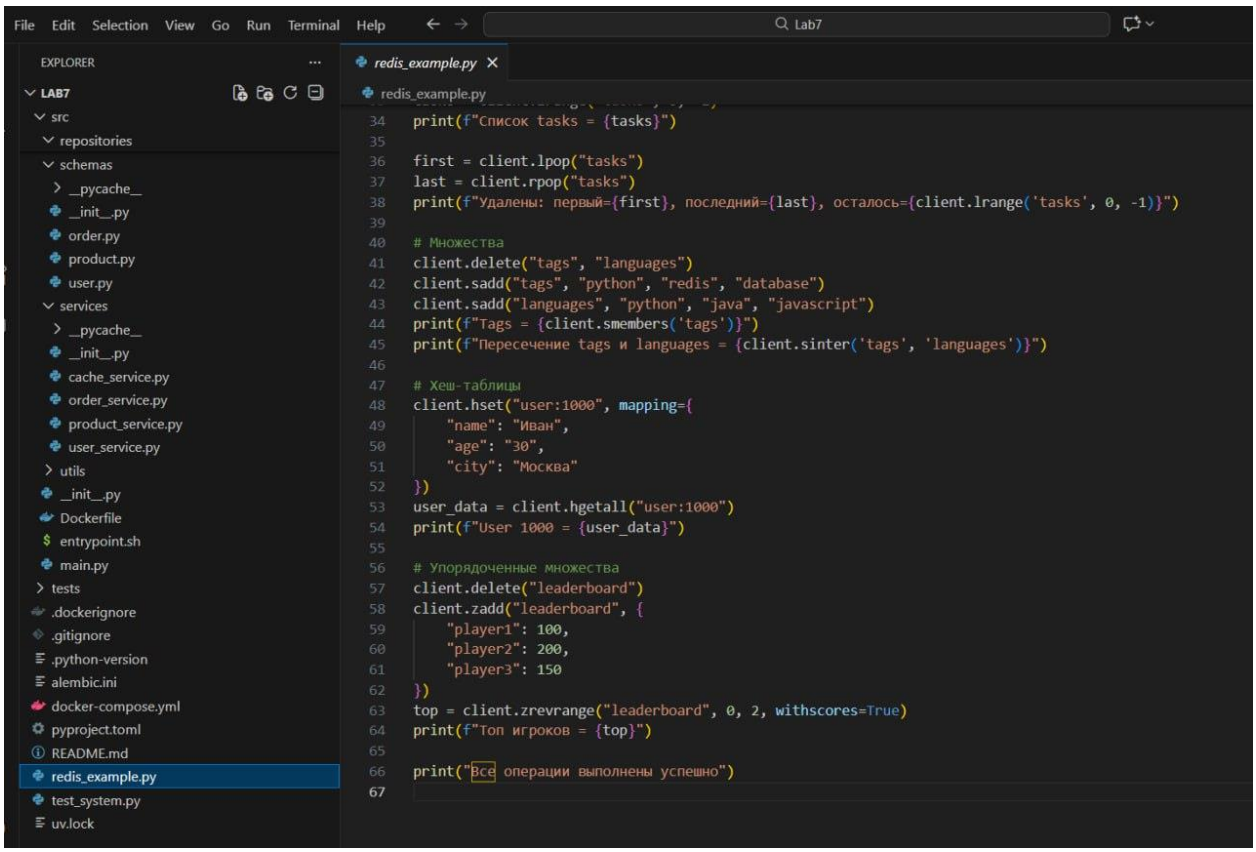
```
async def provide_cache_service() -> CacheService:
    """Провайдер сервиса кэширования."""
     return CacheService(redis_client)
```

- 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)





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

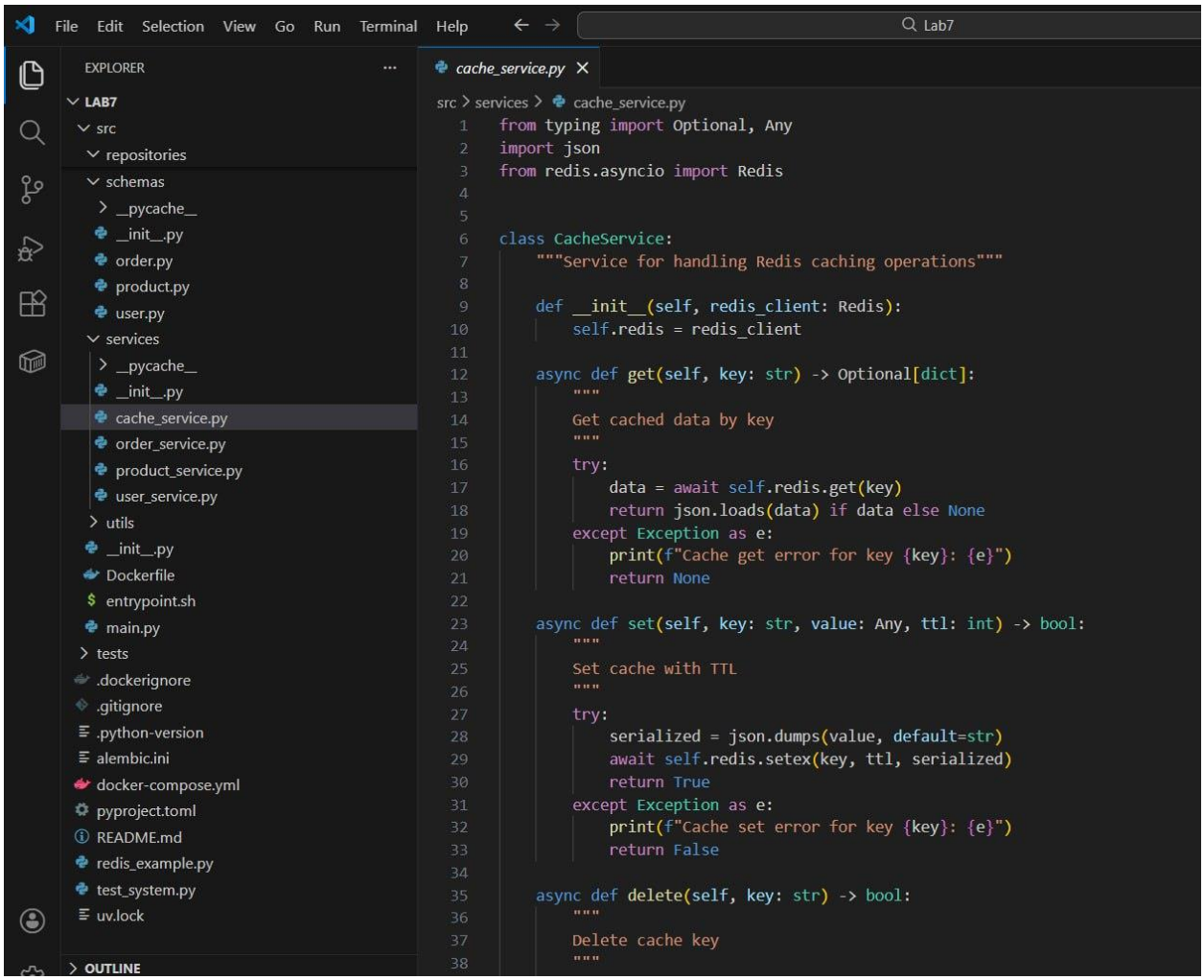
1. Строки - базовое хранилище ключ-значение
2. TTL - автоматическое удаление ключей через время
3. Счетчики - атомарные операции инкремента/декремента
4. Списки - упорядоченные коллекции с быстрым доступом к концам
5. Множества - уникальные элементы с операциями пересечения
6. Хеш-таблицы - структурированное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

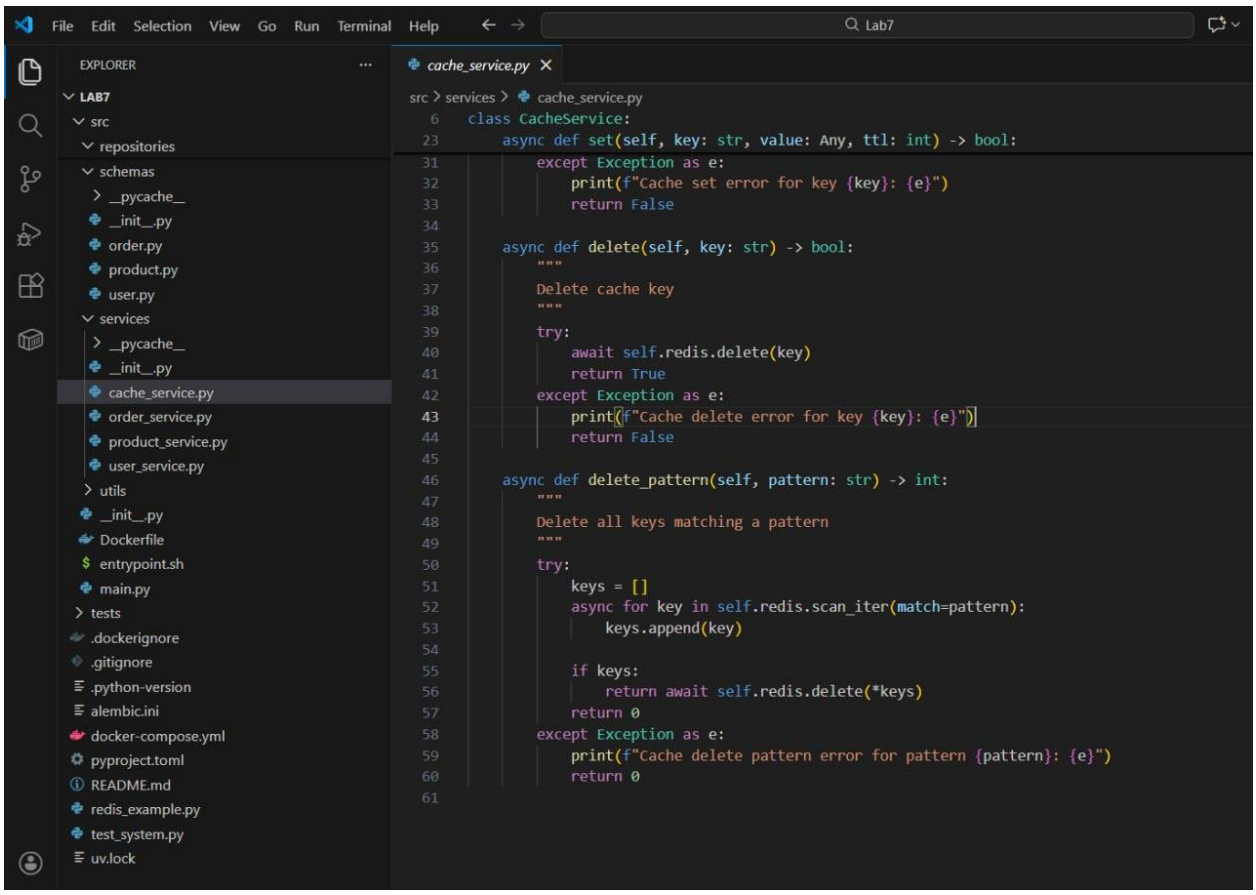
Пример запуска теста:

Часть 4: Кэширование запросов и очищение запросов

Файл: *cache_service.py*

(Создание сервиса кэширования)





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)

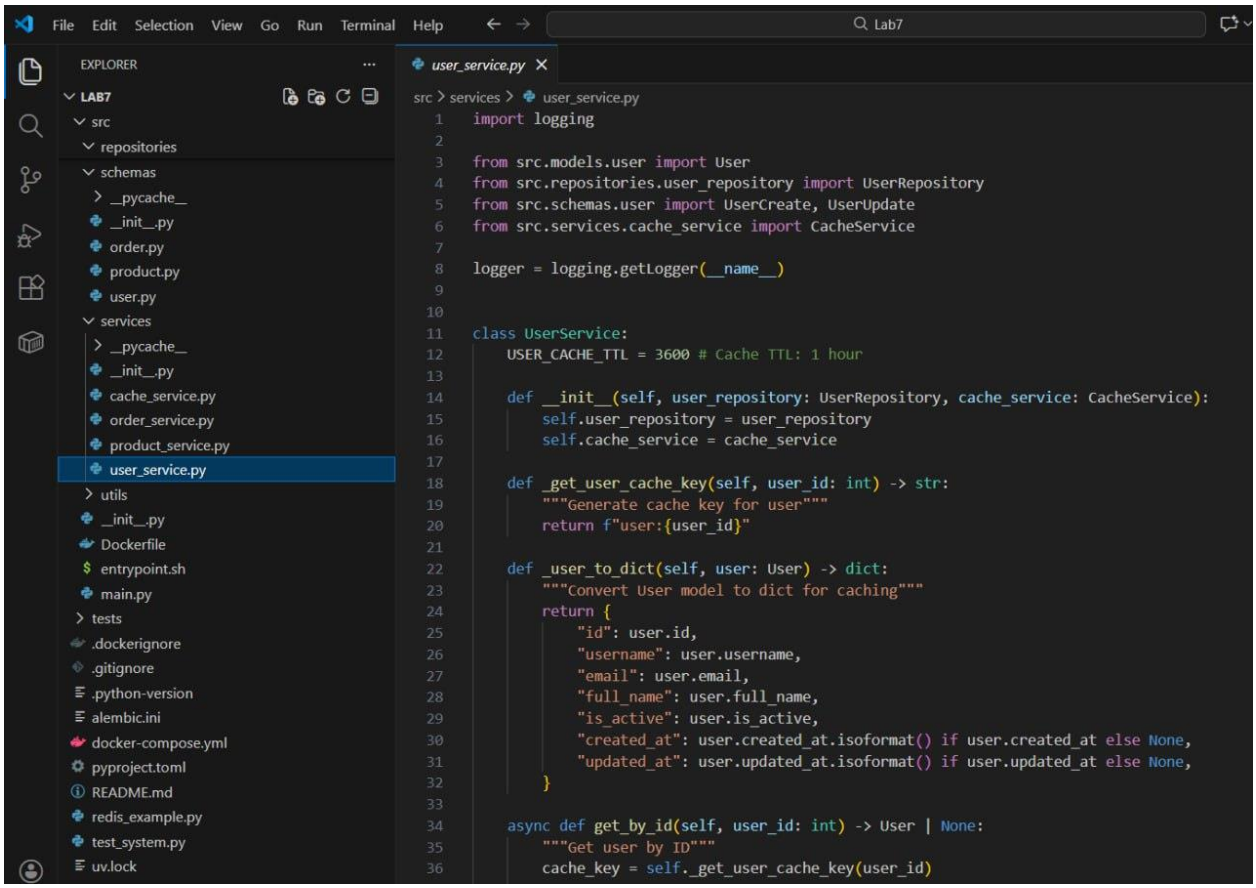
- **set(key, value, ttl)** - сохранение в кэш с автоматическим истечением.
    - json.dumps(value, default=str) - сериализует любые объекты
    - default=str обрабатывает datetime и другие не-JSON типы
    - setex(key, ttl, value) - устанавливает значение с автоматическим истечением
    - ttl в секундах: Redis сам удалит ключ через указанное время

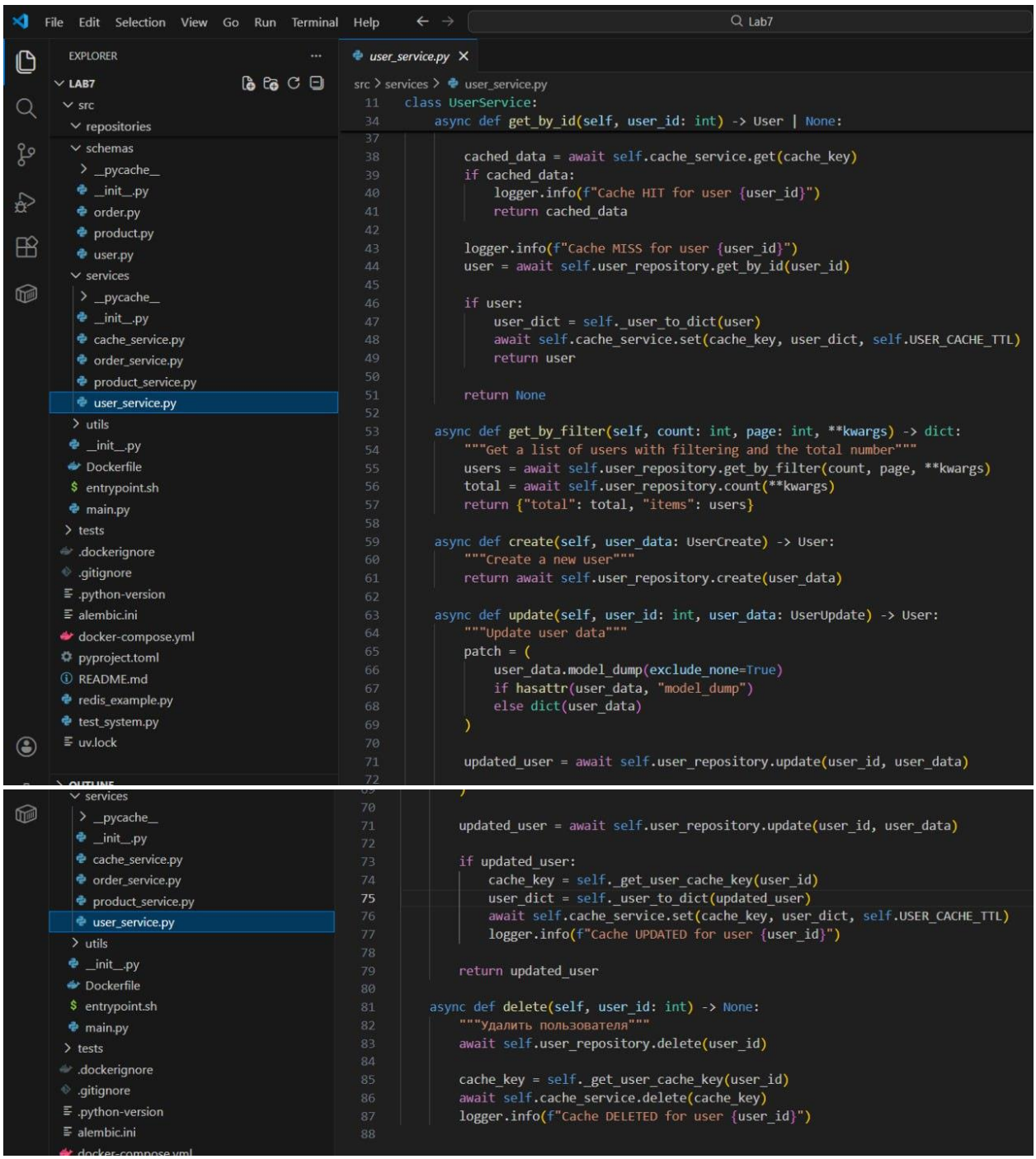
- **delete(key)** - удаление одного ключа
- **delete_pattern(pattern)** - массовое удаление по шаблону.
  scan_iter(match=pattern) - итератор по ключам, не блокирует Redis при большом количестве ключей.

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))





Добавлено кэширование с требуемым временем жизни:

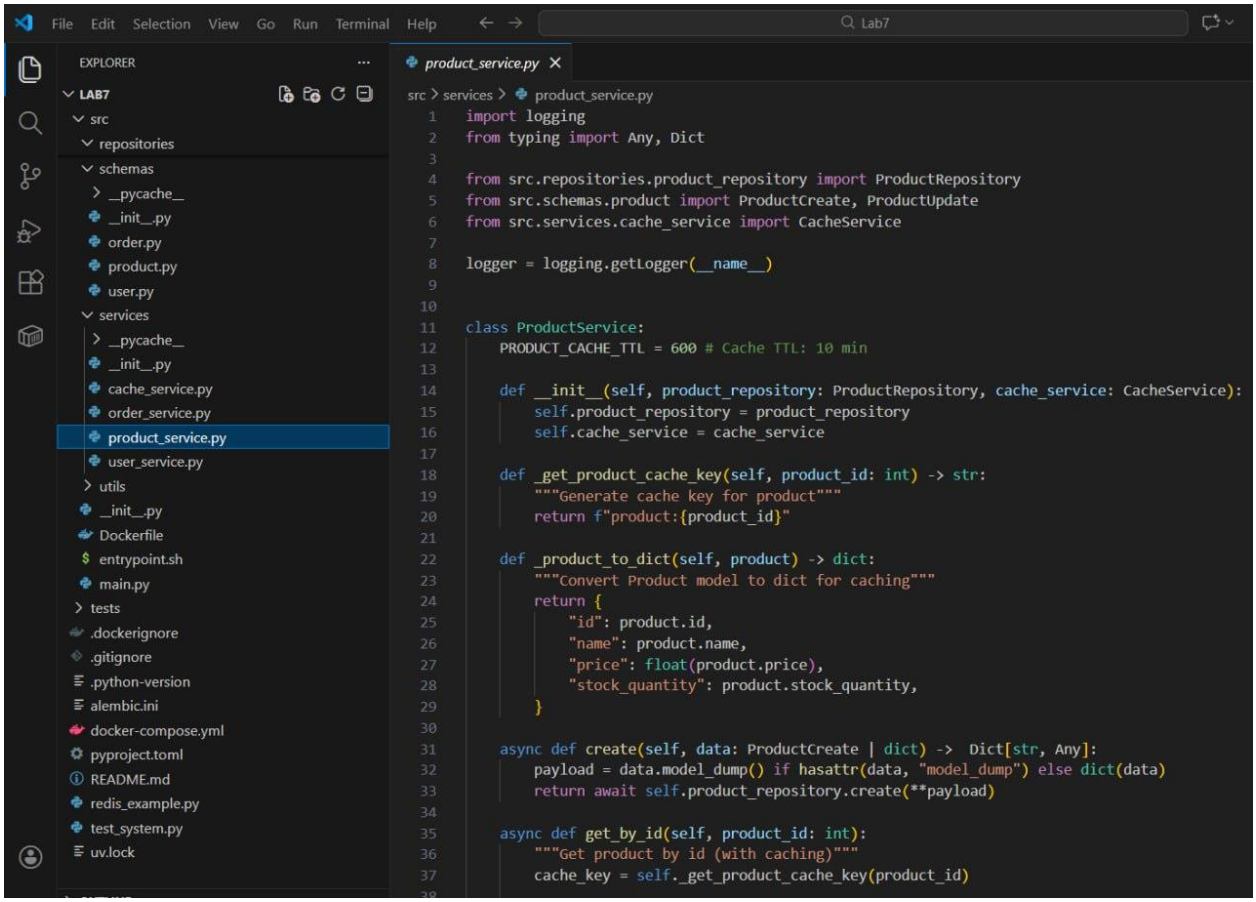
- USER_CACHE_TTL = 3600 (1 час = 3600 секунд)

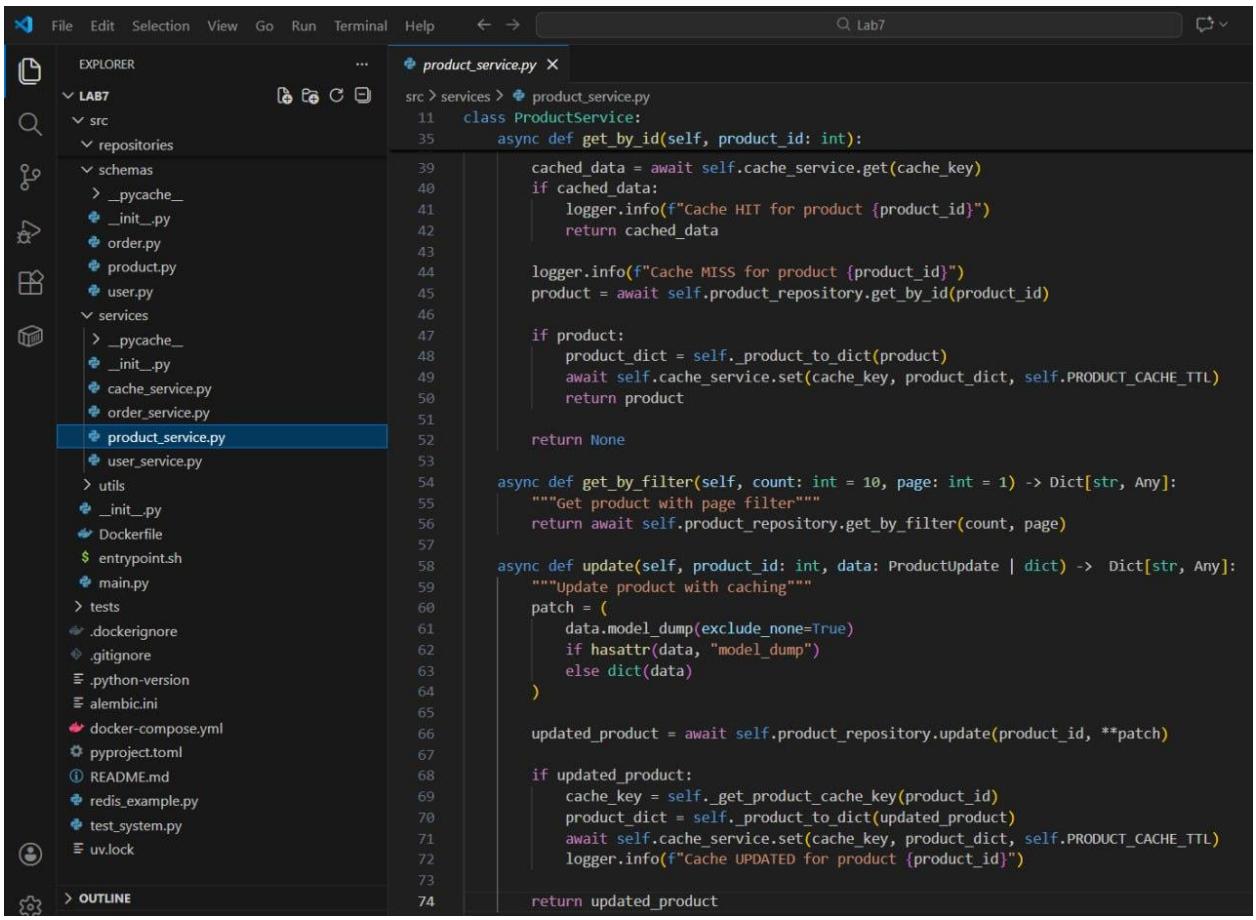
Вспомогательные методы:

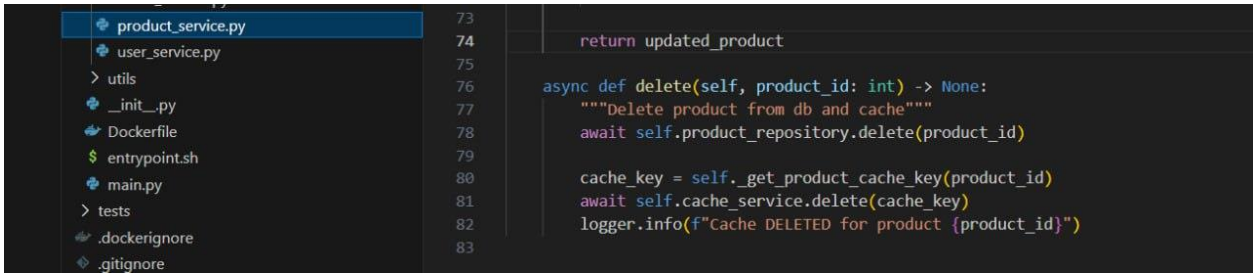
- get_user_cache_key() - генерация ключа кэша
- user_to_dict() - конвертация модели в словарь для кэширования. Необходимо конвертировать в dict т.к. SQLAlchemy модели нельзя напрямую сериализовать в JSON, и таким образом мы можем контролировать, какие поля попадают в кэш.

- 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)**)**







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 кеширования, 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 указную 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 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.