Тестовое задание: REST API для LRU Cache с TTL

Описание:

Необходимо разработать REST API для реализации Least Recently Used (LRU) cache с поддержкой Time-To-Live (TTL).

LRU cache - это структура данных, которая позволяет хранить ограниченное количество элементов, удаляя наименее используемые элементы при достижении лимита или истечении TTL.

Требования:

1. **Реализация API:**
    - **Эндпоинт для получения значения по ключу (GET):**
        - `/cache/{key}`
        - Возвращает значение, связанное с ключом `key`.
        - Если ключ не найден или TTL истек, возвращает код состояния `404 Not Found` и JSON с сообщением об ошибке.
    - **Эндпоинт для добавления/обновления значения по ключу (PUT):**
        - `/cache/{key}`
        - Принимает JSON в теле запроса с ключом `value` (например, `{"value": "some_value"}` и опциональным ключом `ttl` (в секундах) например `{"value": "some_value", "ttl": 60}`).
        - Добавляет или обновляет значение для ключа `key`.
        - Если `ttl` не указан, значение кэшируется без TTL (т.е. на неопределенный срок, пока не будет вытеснено LRU).
        - Возвращает код состояния `200 OK` или `201 Created` (если ключ был добавлен).
    - **Эндпоинт для удаления значения по ключу (DELETE):**
        - `/cache/{key}`
        - Удаляет значение, связанное с ключом `key`.
        - Возвращает код состояния `204 No Content` (успешное удаление) или `404 Not Found` (если ключ не найден).
    - **Эндпоинт для получения статистики cache (GET):**
        - `/cache/stats`
        - Возвращает JSON с информацией о cache, включая:
            - `size`: Текущий размер cache (количество элементов).

- `capacity` : Максимальная емкость cache.
- `items` : Список ключей, находящихся в кэше в порядке LRU (от наиболее до наименее используемого).

2. **Реализация LRU Cache с TTL:**

   - Необходимо реализовать логику LRU cache. Можно использовать встроенные структуры данных Python, такие как `collections.OrderedDict` или `collections.deque` , или реализовать свою структуру.
   - Cache должен иметь ограниченную емкость, задаваемую при инициализации.
   - При попытке добавить новый элемент в полный cache, наименее используемый элемент должен быть удален.
   - Каждое обращение к ключу (GET, PUT, DELETE) должно обновлять его позицию в cache, чтобы он считался наиболее используемым.
   - Каждый элемент в кэше должен иметь опциональное TTL (Time-To-Live).
   - При получении элемента из кэша необходимо проверять, не истек ли его TTL. Если истек, элемент должен быть удален из кэша и возвращен ответ `404` .
   - Не должно быть гонок данных при одновременном доступе к кэшу из разных потоков/процессов (asyncio safety)

3. **Валидация:**

   - Capacity cache и TTL должны быть положительными числами.
   - API должен возвращать ошибки валидации (HTTP 422) если данные не валидны.
   - Должен быть реализован middleware, который логирует каждый запрос и время его обработки.

4. **Тесты:**

   - Необходимо написать unit-тесты для всех эндпоинтов API и логики LRU cache.
   - Тесты должны покрывать основные сценарии использования, включая:
     - Добавление, получение, обновление и удаление элементов.
     - Достижение лимита емкости cache и вытеснение элементов.
     - Обработку несуществующих ключей.
     - Проверку статистики cache.
     - Тестирование TTL: добавление элементов с TTL, проверка истечения TTL и удаление элементов после истечения TTL.

5. **Docker:**

   - Необходимо создать `Dockerfile` для запуска приложения в контейнере Docker.
   - `Dockerfile` должен содержать все необходимые инструкции для установки зависимостей и запуска приложения.

- Должен быть предоставлен `docker-compose.yml` файл для упрощения запуска сервиса с использованием Docker Compose.

6. **Использование:**
   - Приложение должно быть легко запускаемым и конфигурируемым.
   - Необходимо предоставить инструкции по запуску и тестированию приложения в `README.md`.
   - Capacity cache должен задаваться через переменную окружения.

7. **Рекомендации:**
   - Использовать Poetry как пакетный менеджер

Технологии:

- Python 3.8+
- FastAPI
- pytest
- Docker
- Docker Compose
- asyncio (для потокобезопасности)

Пример структуры проекта:

```
lru-cache-api/
├── api/                    # Версионность апи
|   ├── api.py              # API маршруты (GET, PUT, DELETE, stats)
├── app/                    # Основной код приложения
│   ├── __init__.py         # Инициализация пакета app
│   ├── main.py             # Точка входа: FastAPI, маршруты, запуск
│   ├── cache.py            # LRU Cache с TTL (логика кэширования)
│   ├── config.py           # Конфигурация приложения (переменные окружения)
│   ├── middleware.py       # Middleware (логирование запросов)
│   └── models.py           # Pydantic модели (валидация данных)
├── tests/                  # Unit тесты
│   ├── __init__.py         # Инициализация пакета tests
│   └── test_api.py         # Тесты для API и логики кэша
├── Dockerfile              # Инструкции для сборки Docker образа
├── docker-compose.yml      # Определение сервисов для Docker Compose
├── README.md               # Описание, запуск, тестирование проекта
└── requirements.txt        # Зависимости проекта
```