ORES
Научно-практический электронный журнал
Оригинальные исследования (ОРИС), *ISSN 2222-0402*
www.ores.su
original-research.ru

УДК 004.738.5:004.655.3

ПРИМЕНЕНИЕ REDIS ДЛЯ КЭШИРОВАНИЯ В ВЕБ-ПРИЛОЖЕНИИ: ОЦЕНКА ЭФФЕКТИВНОСТИ

Поваров Максим Константинович,
Студент, Поволжский Государственный Университет Телекоммуникаций и Информатики, г. Самара, Российская Федерация
E-mail: maxim.powarov@mail.ru

Ахметшина Элеонора Газинуровна,
К.т.н., доцент каф. ИРС, Поволжский Государственный Университет Телекоммуникаций и Информатики, г. Самара, Российская Федерация
E-mail: e.ahmetshina@psuti.ru

Аннотация

Актуальность исследования обусловлена ростом нагрузки на веб-сервисы и необходимостью снижать задержку операций чтения без масштабирования основной СУБД. Цель работы — разработать модуль кэширования на Redis для REST API и экспериментально оценить влияние кэша на время отклика и пропускную способность. Методика включает реализацию стратегии cache-aside с инвалидацией ключей при изменении данных, развёртывание стека FastAPI/PostgreSQL/Redis и нагрузочное тестирование эндпоинта GET /api/items при четырёх уровнях нагрузки (1, 10, 50, 100 параллельных запросов). Результаты показали снижение среднего времени отклика в 2-2,5 раза и рост RPS до 2,5 раз при повторяющихся чтениях; показатель p95 также существенно улучшился. Выводы: Redis целесообразно применять в небольших read-heavy веб-приложениях как доступный способ повышения производительности; полученные данные применимы при проектировании учебных информационных систем и прототипов веб-сервисов.

Ключевые слова: Redis; кэширование; веб-приложение; производительность; время отклика; in-memory; REST API; оптимизация.

APPLICATION OF REDIS FOR CACHING IN A WEB APPLICATION: PERFORMANCE EVALUATION

Povarov Maksim Konstantinovich,
Student, Povolzhskiy State University of Telecommunications and Informatics, Samara, Russian Federation
E-mail: maxim.powarov@mail.ru

Akhmetshina Eleonora Gazinurovna,
Candidate of Technical Sciences, Associate Professor, Department of IRS

Povolzhskiy State University of Telecommunications and Informatics, Samara, Russian Federation
E-mail: e.ahmetshina@psuti.ru

ABSTRACT

The relevance of this study is driven by the increasing load on web services and the need to reduce read latency without scaling the underlying DBMS. The objective of this study is to develop a Redis-based caching module for a REST API and to experimentally evaluate the impact of the cache on response time and throughput. The methodology includes the implementation of a cache-aside strategy with key invalidation when data is modified, deployment of a FastAPI/PostgreSQL/Redis stack, and load testing of the GET /api/items endpoint at four load levels (1, 10, 50, and 100 concurrent requests). The results showed a 2-2.5-fold decrease in average response time and an up to 2.5-fold increase in RPS for repeated reads; the p95 index also improved significantly. Conclusions: Redis is practical for small, read-heavy web applications as an affordable way to improve performance. The obtained data are applicable in the design of educational information systems and web service prototypes.

Keywords: Redis; caching; web application; performance; response time; in-memory; REST API; optimization.

Введение

Современные веб-приложения всё чаще проектируются как распределённые системы с REST API или GraphQL-интерфейсом, обслуживающим клиентские, мобильные и сторонние клиенты. По мере роста числа пользователей увеличивается нагрузка на сервер приложений и СУБД: каждый запрос к дисковой базе связан с разбором SQL, построением плана, чтением данных и сериализацией ответа. Для операций чтения, составляющих значительную долю трафика типовых CRUD-сервисов (по разным оценкам 70-90 % в информационных и справочных системах), кэширование в оперативной памяти остаётся одним из наименее затратных способов повышения отзывчивости [1; 3]. Во многих учебных и прототипных проектах производительность изначально не учитывается, а вертикальное масштабирование СУБД на этапе эксплуатации требует времени и экспертизы, тогда как внешний кэш можно подключить без изменения клиентской части [2; 4].

Redis (Remote Dictionary Server) — распространённое in-memory хранилище с поддержкой TTL, атомарных операций, pub/sub, Lua-скриптов и масштабирования через Redis Cluster [2; 7]. В малых информационных системах его применяют для ускорения повторяющихся запросов без замены основной СУБД, однако выигрыш необходимо оценивать количественно: кэш усложняет архитектуру за счёт инвалидации, мониторинга hit rate, отказоустойчивости и согласованности данных [4; 10].

Объект исследования — REST API небольшого веб-приложения на стеке FastAPI/PostgreSQL с типовыми CRUD-операциями. Предмет — влияние Redis-кэширования на время отклика и пропускную способность API при read-heavy нагрузке. Научная новизна — количественная оценка эффекта кэширования на стеке Python/FastAPI при фиксированных сценариях нагрузки, характерных для справочников и каталогов, с измерением среднего времени отклика и 95-го перцентиля (p95). Практическая значимость — результаты и рекомендации применимы при проектировании учебных ИС, дипломных проектов и прототипов веб-сервисов с ограниченными ресурсами, но с требованием приемлемой отзывчивости при одновременной работе нескольких пользователей.

Обзор подходов к кэшированию.

Кэширование — фундаментальный приём оптимизации производительности, основанный на локальности обращений: недавно запрошенные данные с высокой вероятностью будут запрошены повторно [1]. Оно снижает число обращений к медленному уровню хранения за счёт сохранения готовых ответов в быстром хранилище. В многоуровневой архитектуре кэш располагается между сервером приложений и СУБД и позволяет обслуживать повторяющиеся запросы без повторного выполнения SQL-операций, агрегаций и сериализации [2].

В литературе и практике выделяют несколько стратегий размещения кэша [3; 5]:

Cache-aside (lazy loading) — приложение само управляет кэшем: при чтении проверяется ключ в Redis, при промахе выполняется запрос к СУБД, результат записывается в кэш и возвращается клиенту.

Read-through / write-through — загрузку и синхронную запись в кэш и БД делегируют отдельному слою.

Write-behind (write-back) — записи буферизуются в кэше и асинхронно сбрасываются в СУБД, что ускоряет запись, но усложняет обеспечение durability.

Для REST API учебных и малых проектов наиболее распространена cache-aside: она проста в реализации, удобна при отладке и не требует замены драйвера СУБД [3].

Redis хранит данные в оперативной памяти, обеспечивая микросекундные задержки доступа по сравнению с миллисекундными задержками дисковых СУБД [7]. Структуры данных Redis (строки, хеши, списки, множества, sorted sets) позволяют хранить как сериализованные JSON-ответы, так и отдельные поля. Для кэширования HTTP-ответов API наиболее естественно использовать строковые ключи со значениями JSON и TTL, ограничивающим время жизни записи и снижающим риск устаревания данных [10].

Критически важен аспект согласованности: при изменении записей в PostgreSQL кэшированная копия должна быть инвалидирована или обновлена, иначе возможен stale read. Для этого применяют комбинацию TTL и явного удаления ключей при POST, PUT и DELETE [4]. Паттерн cache stampede protection в работе не реализовывался, но учитывается как направление развития [2; 10].

Сравнение с Memcached, Hazelcast и встроенным кэшем приложения показывает, что Redis выигрывает за счёт богатого набора структур данных, persistence (RDB/AOF), репликации и экосистемы мониторинга [7]. Memcached проще на чистых get/set, но беднее по возможностям. Встроенный кэш процесса не разделяется между worker-процессами uvicorn/gunicorn, поэтому при горизонтальном масштабировании предпочтителен Redis [2; 8].

Постановка задачи.

Цель работы — разработать модуль кэширования на Redis для REST API и экспериментально оценить влияние кэша на время отклика и пропускную способность при типовой read-heavy нагрузке.

Для достижения поставленной цели сформулированы следующие задачи исследования:

Проанализировать подходы к кэшированию в веб-приложениях и обосновать выбор стратегии cache-aside для REST API на стеке FastAPI/PostgreSQL.

Реализовать модуль кэширования с инвалидацией ключей при операциях записи (POST, PUT, DELETE) и настраиваемым TTL.

Интегрировать Redis в веб-приложение, обеспечив возможность включения и отключения кэша через переменные окружения без изменения клиентского кода.

Разработать скрипт нагрузочного тестирования, воспроизводящий сценарии чтения списка записей при четырёх уровнях параллельной нагрузки.

Сравнить среднее время отклика, 95-й перцентиль (p95) и RPS (requests per second) в режимах with_cache и without_cache.

Проанализировать полученные результаты, выявить закономерности роста задержки с увеличением нагрузки и сформулировать рекомендации по применению Redis в read-heavy веб-сервисах.

Гипотеза исследования: включение Redis-кэша для эндпоинта чтения списка записей существенно снизит время отклика и повысит пропускную способность API при повторяющихся запросах, при этом накладные расходы на сериализацию и сетевое взаимодействие с Redis будут меньше, чем затраты на повторное выполнение SQL-запроса к PostgreSQL.

Архитектура решения.

Прототип построен по трёхзвенной схеме «клиент — сервер приложений — хранилище данных» с дополнительным слоем кэша между сервером и СУБД. Серверная часть реализована на Python 3 (FastAPI), персистентное хранение — PostgreSQL 16, кэш — Redis 7 в контейнере Docker. Разделение компонентов соответствует типовому контуру веб-сервисов и позволяет независимо масштабировать и заменять их [2; 8].

В модуле cache.py реализованы три группы функций: чтение (get_cached), запись (set_cached) и инвалидация (invalidate_prefix). Именование ключей следует соглашению «пространство_имён:тип:идентификатор»: items:list — полный список записей в JSON; items:{id} — запись по идентификатору; items:count — результат COUNT. Префикс items: обеспечивает группировку и упрощает массовую инвалидацию [6; 9].

При операциях POST, PUT и DELETE вызывается invalidate_prefix("items:"), удаляющая все ключи с данным префиксом. Это гарантирует согласованность: после модификации следующий запрос на чтение выполнит cache miss и загрузит актуальные данные из PostgreSQL [4; 6]. Точечная инвалидация снижает нагрузку на Redis, но сложнее в сопровождении; для учебного прототипа выбрана полная инвалидация по префиксу.

TTL по умолчанию — 120 с (переменная CACHE_TTL) и служит страховкой от рассогласования при сбоях или race condition [10]. Режим кэширования включается флагом CACHE_ENABLED=1; при CACHE_ENABLED=0 приложение работает напрямую с PostgreSQL (режим without_cache в эксперименте).

Сериализация выполняется модулем json: ответ кодируется в UTF-8 и сохраняется в Redis как string; при чтении выполняется обратное преобразование. Контракт API не меняется, схема прозрачна для клиента [8].

Развёртывание — через docker-compose: контейнеры postgres и redis поднимаются одной командой, приложение подключается к ним по именам сервисов. Это обеспечивает воспроизводимость эксперимента [4; 11].

При недоступности Redis реализован graceful degradation: исключение перехватывается, выполняется запрос к PostgreSQL. Отказ кэша не делает API недоступным, а лишь временно снижает производительность до уровня without_cache [4; 10].

Методика эксперимента.

Эксперимент выполнялся на веб-приложении с подключённой PostgreSQL и опционально Redis. Тестировался эндпоинт GET /api/items, возвращающий полный список записей в формате JSON. Выбор именно операции чтения списка обусловлен тем, что она наиболее нагружает систему: выполняется SELECT без фильтра по первичному ключу, возвращается полная выборка из 300 записей, результат сериализуется в JSON объёмом нескольких десятков килобайт. Такой сценарий типичен для главных страниц справочников, лент новостей и списков задач [1; 5].

Перед каждым прогоном база данных предзаполнялась 300 однотипными записями через служебный скрипт seed. Объём 300 записей выбран как компромисс между

реалистичностью (список достаточно большой, чтобы SQL-запрос и сериализация занимали измеримое время) и возможностью воспроизведения на обычной рабочей станции без специализированного серверного оборудования. В режиме with_cache перед основным прогоном выполнялся прогрев кэша — 20 последовательных запросов к API, чтобы первый «холодный» промах не искажал статистику основной серии измерений [2; 10].

Параметры нагрузочного тестирования приведены в таблице 1. Скрипт benchmark_redis.py реализован на Python с использованием asyncio и httpx для асинхронных HTTP-запросов. Скрипт выполняет два режима: without_cache (CACHE_ENABLED=0, прямое чтение из PostgreSQL) и with_cache (CACHE_ENABLED=1, чтение через Redis). Для каждого режима и каждого уровня нагрузки фиксировались: среднее время отклика (avg, мс), 95-й перцентиль (p95, мс) и пропускная способность (RPS — число успешно обработанных запросов в секунду).

Таблица 1. Параметры нагрузочного эксперимента

Параметр	Значение
Эндпоинт	GET /api/items
Предзаполнение	300 записей
Прогрев кэша	20 запросов (режим with_cache)
Запросов в прогоне	200
Уровни нагрузки	1, 10, 50, 100
Метрики	avg, p95, RPS

Уровни нагрузки 1, 10, 50 и 100 соответствуют числу одновременных «виртуальных пользователей» — параллельных asyncio-задач, каждая из которых выполняет серию запросов к API. Уровень 1 моделирует последовательную работу одного клиента; уровень 100 — пиковую нагрузку, характерную для одновременного открытия страницы группой пользователей в учебной аудитории или при демонстрации проекта на защите. Промежуточные уровни 10 и 50 позволяют проследить нелинейный рост задержки по мере насыщения ресурсов сервера [2].

Измерения проводились на локальной машине под управлением Linux с SSD-накопителем. Каждый прогон включал 200 запросов к API; между режимами with_cache и without_cache сервер приложений перезапускался для исключения влияния предыдущего состояния — прогретых буферов PostgreSQL, установленных TCP-соединений и остаточных данных в памяти процесса. Дополнительно через команду INFO keyspace в redis-cli фиксировались показатели keyspace_hits и keyspace_misses для подтверждения работы кэша и расчёта hit rate в режиме with_cache.

Методика измерения времени отклика включает полный цикл обработки запроса на стороне клиента: от отправки HTTP GET до получения полного тела ответа. Таким образом, в метрику входят задержки сетевого стека localhost, обработки в FastAPI, обращения к Redis или PostgreSQL, сериализации JSON и передачи данных. Это соответствует воспринимаемому пользователем времени загрузки и является более практичным показателем, чем изолированное время выполнения SQL-запроса [5; 12].

Для обеспечения сопоставимости прогонов фиксировались версии используемого программного обеспечения: Python 3.14, FastAPI 0.115, SQLAlchemy 2.0, драйвер psycopg 3, клиент redis-py 5.x. Сервер приложений запускался через uvicorn с одним worker-процессом, чтобы исключить влияние конкуренции между процессами за пул соединений

и кэш. PostgreSQL и Redis работали в контейнерах Docker с лимитами ресурсов по умолчанию, что соответствует типичной конфигурации учебного стенда без выделенного серверного оборудования [4; 11].

Результаты.

Результаты сравнения режимов с кэшем и без кэша представлены в таблице 2. Зависимость среднего времени отклика от уровня нагрузки иллюстрируется на рисунке 1.

Таблица 2. Влияние Redis-кэширования на время отклика API

Режим	Уровень нагрузки	Среднее, мс	p95, мс	RPS
without_cache	1	8.91	18.14	112.2
with_cache	1	4.39	5.26	227.8
without_cache	10	15.01	24.85	66.6
with_cache	10	6.26	7.86	159.6
without_cache	50	21.09	30.45	47.4
with_cache	50	7.63	15.43	131.1
without_cache	100	27.28	36.47	36.7
with_cache	100	10.90	18.03	91.7

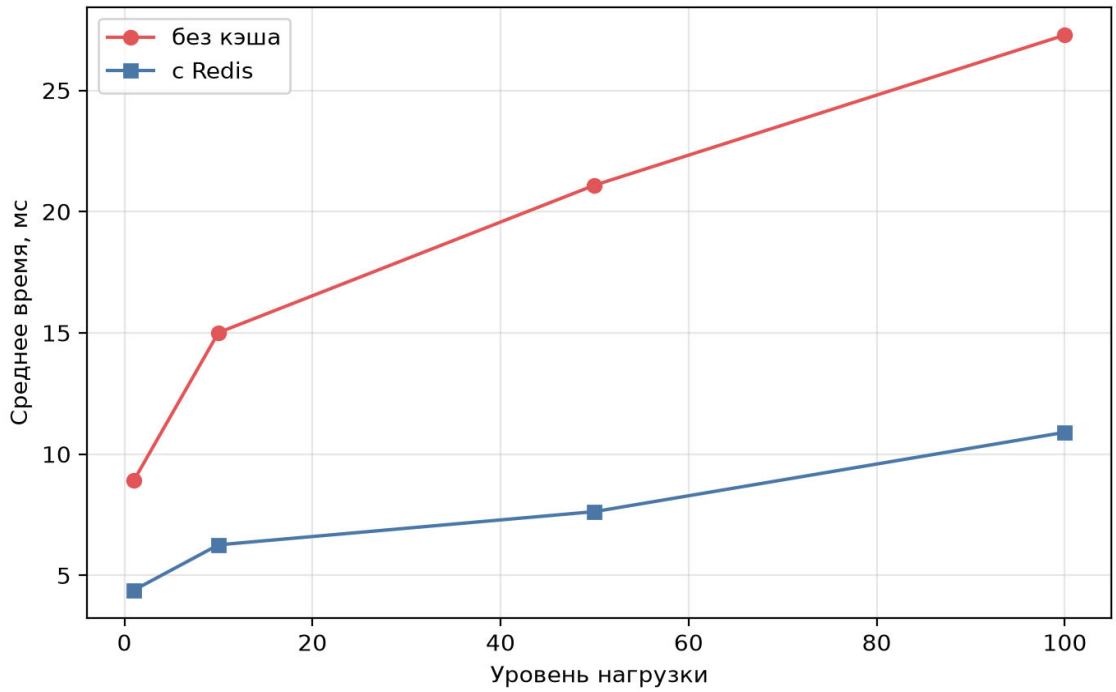

Рисунок 1. Сравнение времени отклика API с кэшем и без кэша.

Анализ результатов при минимальной нагрузке (уровень 1). Уже при последовательной работе одного клиента включение Redis снижает среднее время отклика с 8,91 до 4,39 мс — почти в 2 раза. RPS возрастает с 112,2 до 227,8. Показатель p95 улучшается с 18,14 до 5,26 мс, что означает существенное сокращение «хвоста» распределения задержек даже при отсутствии конкуренции за ресурсы. Выигрыш объясняется тем, что при cache hit приложение не выполняет SQL-запрос к PostgreSQL, не обращается к дисковому подсистеме и не выполняет повторную сериализацию выборки из ORM-объектов — готовый JSON считывается из Redis за микросекунды [2; 7].

Анализ результатов при умеренной нагрузке (уровни 10 и 50). С ростом числа параллельных запросов преимущество кэша сохраняется и усиливается в относительном выражении. При нагрузке 10 среднее время без кэша вырастает до 15,01 мс (рост на 68 % относительно уровня 1), тогда как с кэшем — до 6,26 мс (рост на 43 %). При нагрузке 50 разрыв ещё больше: 21,09 мс против 7,63 мс. RPS без кэша падает до 47,4, с кэшем сохраняется на уровне 131,1 — почти трёхкратное преимущество. Это свидетельствует о том, что PostgreSQL становится узким местом при конкурентных чтениях: несколько параллельных SELECT-запросов конкурируют за соединения из пула, блокировки и CPU на сериализацию, тогда как Redis обслуживает параллельные GET-запросы значительно эффективнее [6; 9].

Анализ результатов при высокой нагрузке (уровень 100). Наибольший абсолютный выигрыш наблюдается при максимальной нагрузке: средняя задержка снижается с 27,28 до 10,90 мс, p95 — с 36,47 до 18,03 мс, RPS возрастает с 36,7 до 91,7 (в 2,5 раза). График на рисунке 1 наглядно показывает расхождение кривых: без кэша время отклика растёт быстрее с увеличением нагрузки, формируя более крутой наклон. С кэшем кривая остаётся более пологой, что указывает на лучшую масштабируемость read-heavy сценария при наличии прогретого кэша [2; 3].

Сводная оценка. При включённом кэше среднее время отклика снижается в 2-2,5 раза на всех уровнях нагрузки, RPS возрастает до 2,5 раз за счёт исключения повторных SQL-запросов и сериализации больших выборок. Показатель p95 улучшается существенно на каждом уровне, что важно для пользовательского опыта: именно «хвост» распределения задержек определяет воспринимаемую отзывчивость интерфейса при пиковых обращениях к API, когда часть запросов «застревает» дольше среднего [1; 12]. Коэффициент ускорения p95 при нагрузке 100 составляет 2,02 (36,47 / 18,03), что подтверждает равномерное улучшение не только средних, но и «худших» задержек.

Таблица 3. Коэффициент ускорения при включении Redis-кэша

Уровень нагрузки	Ускорение avg	Ускорение p95	Ускорение RPS
1	2,03	3,45	2,03
10	2,40	3,16	2,40
50	2,76	1,97	2,77
100	2,50	2,02	2,50

Как видно из таблицы 3, коэффициент ускорения по среднему времени отклика монотонно растёт от 2,03 при нагрузке 1 до 2,76 при нагрузке 50, несколько снижаясь до 2,50 при нагрузке 100. Такая динамика объясняется тем, что при росте параллелизма PostgreSQL быстрее насыщается, тогда как Redis продолжает эффективно обслуживать параллельные чтения. Ускорение p95 на уровне 1 достигает 3,45 — наибольшего значения среди всех уровней, — поскольку при последовательных запросах «холодные» обращения к БД создают более длинный хвост распределения, чем стабильные cache hits из Redis [2; 7].

Обсуждение.

Полученные результаты согласуются с теоретическими представлениями о роли кэширования в многоуровневых архитектурах [1; 2]. Кэширование наиболее эффективно для read-heavy сценариев: каталоги, справочники, ленты заданий, результаты мониторинга с редким обновлением [1; 5]. В рассмотренном приложении преобладает чтение списка, что соответствует типичному профилю нагрузки ИС вуза, где одни и те же данные просматривают многократно между редкими операциями изменения.

Для write-heavy систем (бронирование, финансовые транзакции, совместное редактирование) выигрыш ограничен частой инвалидацией при каждой записи [4]. Тем не менее даже при соотношении чтений и записей 80/20 кэш может давать существенный эффект, если окно между записью и массовым чтением достаточно велико для накопления cache hits [2; 10].

Внедрение Redis оправдано, когда стоимость медленного отклика выше стоимости поддержки дополнительного сервиса. В учебном проекте Redis в Docker поднимается командой docker compose up и обеспечивает воспроизводимость эксперимента. Для промышленной эксплуатации рекомендуется: кэшировать часто читаемые и редко изменяемые данные; задавать TTL даже при ручной инвалидации; мониторить hit rate через INFO stats; реализовать graceful degradation при недоступности Redis [2; 10; 11].

Альтернативы без кэша — индексы в PostgreSQL, материализованные представления и увеличение числа worker-процессов uvicorn — полезны, но не устраняют накладные расходы сериализации и сети либо увеличивают нагрузку на пул соединений. Redis-кэш дополняет эти меры и может применяться совместно с ними [6; 9].

Эксперимент выполнен на одном узле: не исследованы Redis Cluster, Sentinel, cache stampede и поведение при отказе кэш-сервера. Не измерено влияние кэширования на GET /api/items/{id} и GET /api/items/count. Дальнейшая работа может включить тестирование на нескольких узлах с общим Redis, оценку влияния размера выборки (100, 1000, 10 000 записей) и сравнение JSON с MessagePack или Protocol Buffers [4; 11].

Заключение

В работе разработан и интегрирован модуль Redis-кэширования в веб-приложение с REST API на стеке FastAPI/PostgreSQL. Реализована стратегия cache-aside с инвалидацией ключей по префиксу при записи, настраиваемым TTL и включением/отключением кэша через переменные окружения. Проведено нагрузочное тестирование эндпоинта GET /api/items при нагрузке 1, 10, 50 и 100 параллельных запросов в режимах with_cache и without_cache.

Экспериментально подтверждено снижение времени отклика при повторяющихся чтениях в среднем в 2-2,5 раза (таблица 2, рисунок 1). Пропускная способность возросла до 2,5 раз; улучшился и 95-й перцентиль задержки. Наибольший эффект — при 100 параллельных запросах: задержка снизилась с 27,28 до 10,90 мс, RPS — с 36,7 до 91,7.

Redis целесообразно применять в небольших веб-приложениях с преобладанием операций чтения как доступный способ повышения производительности без масштабирования СУБД и изменения клиентской части. Результаты применимы при проектировании учебных ИС, дипломных проектов и прототипов веб-сервисов с ограниченными ресурсами и повторяющимися запросами к одним и тем же данным.

Список литературы:

1. Фаулер, М. Архитектура корпоративных программных приложений / М. Фаулер ; пер. с англ. — Москва : Вильямс, 2006. — 544 с. — ISBN 978-5-8459-0579-6.
2. Клеппман, М. Высоконагруженные приложения. Программирование, масштабирование, поддержка / М. Клеппман ; пер. с англ. — Санкт-Петербург : Питер, 2018. — 640 с. — ISBN 978-5-4461-0512-0.
3. Ричардсон, К. Микросервисы. Паттерны разработки и рефакторинга / К. Ричардсон ; пер. с англ. — Санкт-Петербург : Питер, 2019. — 544 с. — ISBN 978-5-4461-0996-8.
4. Нейгард, М. Release it! Проектирование и дизайн ПО для тех, кому не всё равно / М. Нейгард ; пер. с англ. — Санкт-Петербург : Питер, 2016. — 320 с. — ISBN 978-5-496-01611-7.

5. Гагарина, Л. Г. Технология разработки программного обеспечения : учеб. пособие / Л. Г. Гагарина, Е. В. Кокорева, Б. Д. Сидорова-Виснадул ; под ред. Л. Г. Гагариной. — Москва : ИД «Форум» : ИНФРА-М, 2018. — 400 с. — ISBN 978-5-8199-0812-9.
6. Рогов, Е. В. PostgreSQL 16 изнутри / Е. В. Рогов. — Москва : ДМК Пресс, 2024. — 664 с. — ISBN 978-5-93700-305-8.
7. Документация Redis [Электронный ресурс]. — Режим доступа: https://redis.io/docs/latest/ (дата обращения: 27.06.2026).
8. Документация FastAPI [Электронный ресурс]. — Режим доступа: https://fastapi.tiangolo.com/ (дата обращения: 27.06.2026).
9. Документация PostgreSQL 16 [Электронный ресурс] / PostgreSQL Global Development Group. — Режим доступа: https://www.postgresql.org/docs/16/ (дата обращения: 27.06.2026).
10. Optimizing Redis [Электронный ресурс] // Redis Documentation. — Режим доступа: https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/ (дата обращения: 27.06.2026).
11. Таненбаум, Э. Современные операционные системы / Э. Таненбаум, Х. Бос ; пер. с англ. — 4-е изд. — Санкт-Петербург : Питер, 2015. — 1120 с. — ISBN 978-5-496-01395-6.
12. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин ; пер. с англ. — Санкт-Петербург : Питер, 2018. — 352 с. — ISBN 978-5-4461-0772-8.

References:

1. Fowler, M. Architecture of Enterprise Software Applications / M. Fowler; translated from English. — Moscow: Williams, 2006. — 544 p. — ISBN 978-5-8459-0579-6.
2. Kleppman, M. Highly Loaded Applications. Programming, Scaling, Support / M. Kleppman; translated from English. — St. Petersburg: Piter, 2018. — 640 p. — ISBN 978-5-4461-0512-0.
3. Richardson, K. Microservices. Development and Refactoring Patterns / K. Richardson; translated from English. — St. Petersburg: Piter, 2019. — 544 p. — ISBN 978-5-4461-0996-8.
4. Neigard, M. Release it! Software Engineering and Design for Those Who Care / M. Neigard; translated from English. — Saint Petersburg: Piter, 2016. — 320 p. — ISBN 978-5-496-01611-7.
5. Gagarina, L. G. Software Development Technology: a textbook / L. G. Gagarina, E. V. Kokoreva, B. D. Sidorova-Visnadul; edited by L. G. Gagarina. — Moscow: Forum Publishing House: INFRA-M, 2018. — 400 p. — ISBN 978-5-8199-0812-9.
6. Rogov, E. V. PostgreSQL 16 from the inside / E. V. Rogov. — Moscow: DMK Press, 2024. — 664 p. — ISBN 978-5-93700-305-8.
7. Redis Documentation [Electronic resource]. — Access mode: https://redis.io/docs/latest/ (accessed date: June 27, 2026).
8. FastAPI Documentation [Electronic resource]. — Access mode: https://fastapi.tiangolo.com/ (accessed date: June 27, 2026).

9. PostgreSQL 16 Documentation [Electronic resource] / PostgreSQL Global Development Group. — Access mode: https://www.postgresql.org/docs/16/ (accessed date: June 27, 2026).
10. Optimizing Redis [Electronic resource] // Redis Documentation. — Available at: https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/ (Accessed: June 27, 2026).
11. Tanenbaum, E. Modern Operating Systems / E. Tanenbaum, H. Bos; translated from English. — 4th ed. — St. Petersburg: Piter, 2015. — 1120 p. — ISBN 978-5-496-01395-6.
12. Martin, R. Clean Architecture: The Art of Software Development / R. Martin; translated from English. — St. Petersburg: Piter, 2018. — 352 p. — ISBN 978-5-4461-0772-8.