Министерство науки и высшего образования РФ  
ФГАОУ ВПО

Национальный исследовательский технологический университет «МИСИС»

Институт компьютерных наук (ИКН)

Кафедра Инфокоммуникационных технологий (ИКТ)

**Курсовая работа**

по дисциплине «Архитектурирование»

Выполнил:  
студент группы БИВТ-22-СП-2

Денисов Д. А.

Оглавление

[Введение 3](#_Toc198440114)

[Цель работы 3](#_Toc198440115)

[Используемые технологии 3](#_Toc198440116)

[Теоретическая часть 5](#_Toc198440117)

[Контейнеризация в современном программировании 5](#_Toc198440118)

[Кеширование и его роль в архитектуре 6](#_Toc198440119)

[Event-driven архитектура 7](#_Toc198440120)

[Проектирование архитектуры 8](#_Toc198440121)

[Общая архитектура 8](#_Toc198440122)

[Функционал приложения 10](#_Toc198440123)

[Контейнеризация приложения 10](#_Toc198440124)

[Стратегия кеширования 11](#_Toc198440125)

[Бизнес логика 11](#_Toc198440126)

[Функционал модели Producer/Consumer 12](#_Toc198440127)

[Реализация 13](#_Toc198440128)

[Структура проекта 13](#_Toc198440129)

[Тестирование 15](#_Toc198440130)

[Кеширование 15](#_Toc198440131)

[RabbitMQ 16](#_Toc198440132)

[Заключение 17](#_Toc198440133)

[Список литературы 19](#_Toc198440134)

# Введение

В условиях стремительного развития информационных технологий разработка программного обеспечения требует использования современных архитектурных подходов, обеспечивающих масштабируемость, надёжность и удобство развертывания. Особенно актуальными становятся решения, позволяющие эффективно управлять взаимодействием между компонентами системы, а также изолировать их для повышения отказоустойчивости и гибкости. В данной курсовой работе рассматривается проектирование и реализация программной системы, основанной на принципах микросервисной архитектуры с использованием таких технологий, как Docker, Redis и RabbitMQ, обеспечивающих высокую производительность и надёжную интеграцию компонентов.

## Цель работы

Цель настоящей работы заключается в разработке архитектуры программного решения, способного обеспечивать эффективную обработку и передачу данных между различными компонентами системы. Для достижения этой цели предполагается использование современных технологий контейнеризации, кэширования и обмена сообщениями. В рамках работы планируется построение архитектурной схемы, обоснование выбора применяемых инструментов, а также демонстрация взаимодействия микросервисов в предложенной архитектуре.

## Используемые технологии

Docker — это современная платформа контейнеризации, предназначенная для создания, упаковки и запуска приложений в изолированных средах, называемых контейнерами. Каждый контейнер содержит всё необходимое для работы приложения — код, зависимости, библиотеки и конфигурации, что обеспечивает стабильность и воспроизводимость в любой среде, будь то локальная разработка или облачная инфраструктура. Особенно эффективен Docker в микросервисной архитектуре, где изоляция компонентов и независимость окружений играют ключевую роль. Он снижает риск конфликта зависимостей, ускоряет процесс развертывания, облегчает автоматизацию с использованием CI/CD и делает масштабирование системы более гибким. Благодаря низкому потреблению ресурсов и высокой скорости запуска, контейнеры являются оптимальным решением для современных программных решений.

Redis — это высокопроизводительное хранилище данных, функционирующее в оперативной памяти и предназначенное для кэширования и временного хранения информации. Благодаря быстрому доступу к данным и поддержке различных структур — таких как строки, списки, множества и хэши — Redis широко применяется в системах, где важна минимальная задержка отклика. Он особенно полезен при работе с пользовательскими сессиями, временными токенами, кэшированием результатов запросов и другими часто используемыми данными. Одной из ключевых возможностей Redis является механизм автоматического удаления устаревших записей, что позволяет эффективно управлять временными данными. В рамках данной работы Redis используется как вспомогательное средство для разгрузки основной базы данных, ускорения обработки запросов и повышения производительности микросервисной системы.

RabbitMQ — это система обмена сообщениями (message broker), обеспечивающая асинхронное взаимодействие между компонентами программной архитектуры посредством очередей. Вместо прямого вызова одного сервиса другим, взаимодействие осуществляется через публикацию сообщений: отправитель помещает данные в очередь, а получатель извлекает их при готовности к обработке. Такой механизм снижает степень связности между микросервисами, повышает гибкость, надёжность и масштабируемость системы. RabbitMQ эффективно справляется с задачами фоновой обработки, маршрутизации сообщений и гарантированной доставки, что делает его незаменимым в распределённых и высоконагруженных системах. В рамках данной курсовой работы RabbitMQ выполняет роль коммуникационного слоя, связывая микросервисы и обеспечивая устойчивую передачу данных в условиях асинхронной обработки.

# Теоретическая часть

## Контейнеризация в современном программировании

Контейнеризация представляет собой технологию, позволяющую упаковать приложение вместе со всеми необходимыми зависимостями, библиотеками и настройками в единый, изолированный контейнер. В отличие от виртуальных машин, контейнеры не содержат собственной операционной системы, а используют ядро хоста, что делает их лёгкими, быстрыми и экономичными в использовании ресурсов.

Основным преимуществом контейнеров является воспроизводимость среды: приложение будет стабильно работать вне зависимости от окружения, будь то локальный компьютер, тестовая среда или облачная платформа. Это свойство особенно ценно при разработке распределённых систем и микросервисной архитектуры, где каждый сервис может разворачиваться отдельно и независимо.

Контейнеризация значительно упростила процессы CI/CD, сделала разработку и сопровождение ПО более гибкими. Наиболее широко используемым инструментом контейнеризации является Docker — он предоставляет всё необходимое для сборки, управления и запуска контейнеров. В связке с Docker Compose можно описывать сложные приложения, состоящие из множества сервисов, в виде конфигурационных файлов и запускать их одной командой. Это ускоряет развертывание, облегчает тестирование и способствует эффективной разработке современных программных решений.

## Кеширование и его роль в архитектуре

Кеширование — это метод повышения производительности информационных систем за счёт хранения часто запрашиваемых данных в быстром хранилище, обычно в оперативной памяти. Такой подход позволяет существенно сократить время обработки запросов и снизить нагрузку на основную базу данных или внешние сервисы, особенно когда одни и те же данные запрашиваются многократно без изменений.

Кеш может быть реализован как на клиентской, так и на серверной стороне. На сервере кеш часто используется для хранения результатов ресурсоёмких операций — например, запросов к базе данных, обращений к API сторонних сервисов или промежуточных расчётов. При повторном запросе результат извлекается из кеша, что позволяет системе отвечать быстрее и обрабатывать больше запросов в единицу времени.

Одним из наиболее распространённых инструментов серверного кеширования является Redis — быстрый, надежный и масштабируемый хранилище данных в оперативной памяти. Он поддерживает различные структуры данных (строки, списки, множества, хэши), что делает его универсальным решением для множества задач. Redis также предлагает дополнительные возможности, такие как автоматическое удаление устаревших данных, система подписки и публикации (Pub/Sub), работа с TTL, а также поддержка репликации и кластеров.

Благодаря низкой задержке при доступе к данным Redis эффективно используется не только для кеширования, но и для хранения сессий, временных токенов, управления ограничениями частоты запросов, реализации очередей и других задач в высоконагруженных и распределённых системах. Его применение особенно актуально в микросервисных архитектурах, где важна быстрая и надёжная передача информации между компонентами.

## Event-driven архитектура

Событийно-ориентированная архитектура (EDA) — это парадигма проектирования программных систем, в которой компоненты взаимодействуют друг с другом посредством генерации и обработки событий. В отличие от традиционного подхода, основанного на синхронных вызовах, EDA предполагает асинхронный обмен сообщениями: один компонент публикует событие, а другие — подписываются на его получение и реагируют на него, когда готовы.

В основе такой архитектуры лежит событие — сообщение, отражающее факт изменения состояния или наступления определённого события в системе, например, «новый заказ создан» или «оплата завершена». Эти события передаются через событийную шину или брокер сообщений, который выступает в роли посредника между отправителем и получателями. Благодаря этому компоненты системы остаются слабо связанными и могут развиваться независимо друг от друга.

Одним из ключевых преимуществ EDA является высокая гибкость и расширяемость. Появление нового бизнес-процесса или функционала можно реализовать через подключение дополнительного обработчика к уже существующим событиям без изменения основного кода. Это особенно важно в условиях быстрой эволюции требований и масштабируемости распределённых систем. Кроме того, асинхронный характер взаимодействия повышает отказоустойчивость — сбои отдельных компонентов не приводят к остановке всей системы, поскольку события могут быть временно сохранены и обработаны позже.

Для технической реализации событийных систем используются брокеры сообщений, такие как RabbitMQ, Kafka или Amazon SQS. Эти платформы обеспечивают доставку, маршрутизацию и хранение сообщений, поддерживают очереди и различные шаблоны обмена информацией. RabbitMQ, например, реализует протокол AMQP и предоставляет гибкие инструменты для построения устойчивой событийной инфраструктуры.

EDA особенно эффективна в высоконагруженных и распределённых приложениях, таких как системы электронной коммерции, аналитические платформы, финансовые сервисы или IoT-решения. Благодаря модульности, масштабируемости и возможности обрабатывать события в реальном времени, событийно-ориентированная архитектура становится важным элементом современных программных решений.

# Проектирование архитектуры

## Общая архитектура

Проект реализован на базе микросервисной архитектуры, что обеспечивает гибкость, модульность и удобство масштабирования. Каждый компонент системы выполняет строго определённую функцию и развернут в отдельном окружении, что упрощает как разработку, так и эксплуатацию.

В качестве основного интерфейса выступает REST API, реализованный на языке Java. Он отвечает за приём внешних запросов, маршрутизацию и первичную обработку данных. Постоянное хранилище построено на базе PostgreSQL — мощной реляционной СУБД, обеспечивающей целостность данных, поддержку транзакций и выполнение сложных SQL-запросов.

Для ускорения доступа к часто запрашиваемым данным в системе используется кеширующий механизм на основе Redis с ограниченным временем жизни записей (TTL). Такой подход позволяет эффективно управлять временным хранением данных, снижать количество обращений к базе данных и повышать общую производительность системы.

Асинхронное взаимодействие между компонентами системы организовано с помощью брокера сообщений RabbitMQ, работающего по модели «поставщик-потребитель». Это позволяет разгрузить основные сервисы, обрабатывать задачи в фоне и обеспечивать устойчивую доставку сообщений даже при временных сбоях.

Все сервисы упакованы в изолированные контейнеры с помощью Docker, а развертывание и взаимодействие между ними управляется через docker-compose. Такой подход гарантирует разделение сред разработки и продакшена, а также облегчает масштабирование и обновление компонентов.

В совокупности предложенная архитектура обеспечивает надёжность, масштабируемость и высокую производительность системы, что делает её устойчивой к росту нагрузки и изменениям в бизнес-логике.

## Функционал приложения

Разработанное приложение реализует ряд функций, обеспечивающих приём, обработку и хранение пользовательских запросов с использованием современных архитектурных подходов:

1. REST API интерфейс:
   * Обрабатывает входящие GET-запросы от пользователей.
   * Возвращает полную историю всех ранее выполненных запросов.
2. Хранение истории запросов в реляционной СУБД PostgreSQL;
3. Кеширование с использованием Redis:
   * При обращении к данным сначала проверяется кеш. При наличии — данные возвращаются немедленно, минуя БД.
4. Асинхронная модель обработки событий по схеме Producer–Consumer через очередь сообщений RabbitMQ.

## Контейнеризация приложения

Для упрощения развертывания в проекте используется Docker. Все ключевые компоненты системы — REST API-сервер, база данных PostgreSQL, кеш Redis и очередь сообщений RabbitMQ — контейнеризованы и запускаются с помощью docker-compose.

Файл docker-compose.yml содержит конфигурации для каждого сервиса, включая настройки сетей, переменных окружения, томов для сохранения данных и открытых портов. Это позволяет легко поднимать все компоненты системы в изолированной среде одной командой, что особенно удобно как для локальной разработки, так и для тестирования и развёртывания на сервере.

## Стратегия кеширования

В приложении реализован механизм кеширования на основе Redis — высокопроизводительного хранилища данных в памяти. Для управления актуальностью данных используется ограничение времени жизни (TTL), после истечения которого записи автоматически удаляются из кеша. Такой подход позволяет эффективно сокращать нагрузку на базу данных, ускорять обработку повторяющихся запросов и оптимизировать использование памяти, обеспечивая поведение, схожее с LRU без необходимости вручную управлять вытеснением элементов.

## Бизнес логика

1. Пользователь отправляет GET-запрос к REST API сервису;
2. Система обращается к кешу (Redis), проверяя наличие данных по заданному идентификатору:
   * Если запись найдена, она немедленно возвращается пользователю;
   * Если запись отсутствует, производится обращение к PostgreSQL для получения данных, после чего результат сохраняется в кеш и отправляется пользователю.
3. Приложение параллельно формирует событие (например, логирование обращения или аналитическая метка) и публикует его в очередь сообщений RabbitMQ;
4. Асинхронный консьюмер, подключённый к очереди, получает это событие и выполняет соответствующие действия — например, сохраняет лог, отправляет уведомление или инициирует внешнюю интеграцию.

## Функционал модели Producer/Consumer

Producer

* Является частью API сервиса;
* Формирует сообщение, описывающее событие, на основе входящего HTTP-запроса;
* Публикует сообщение в соответствующую очередь RabbitMQ.

Consumer

* Представляет собой отдельный сервис, работающий независимо от API;
* Подключён к очереди и непрерывно получает сообщения, поступающие от Producer'а;
* Выполняет необходимые операции на основе полученного события: запись логов, интеграции с внешними системами, обновление данных и прочее.

Ключевые архитектурные особенности

1. Гибкость масштабирования — количество consumer-сервисов можно увеличивать в зависимости от текущей нагрузки;
2. Устойчивость к сбоям: сообщения в очереди сохраняются до момента обработки, обеспечивая надёжную доставку;
3. Поддержка отложенной и параллельной обработки без блокировки API-логики.

# Реализация

Архитектура приложения реализована в соответствии с принципами микросервисного подхода, при котором каждая логическая часть системы выделена в отдельный сервис с четкой зоной ответственности. Все компоненты — API-сервер, база данных, брокер сообщений и кеш — запускаются в изолированных Docker-контейнерах, обеспечивая воспроизводимость среды и удобство развертывания.

В архитектуре используются паттерны **CQRS** (разделение операций чтения и записи) и **Event Sourcing**, позволяющие не только разделять ответственность между слоями, но и отслеживать историю изменений через события. Структура проекта организована по принципу модульности, что упрощает поддержку кода и облегчает масштабирование отдельных компонентов по мере роста нагрузки.

## Структура проекта

**Пользовательский сервис (user-service)**

Сервис предназначен для управления учетными записями пользователей и реализует полный набор CRUD-операций через REST API. Архитектура построена по многослойному принципу, обеспечивая чистое разделение ответственности и расширяемость:

* **Контроллер (UserRestApiController);**Обрабатывает входящие HTTP-запросы и возвращает ответы клиенту. Делегирует выполнение логики в сервисный слой.
* **Сервисный слой (UserService).**Содержит бизнес-логику управления пользователями: создание, получение, обновление и удаление данных. Также отвечает за взаимодействие с кешем и базой данных.

**Кеширование (Redis)**

Используется стратегия cache-aside, при которой данные сначала запрашиваются из Redis. В случае отсутствия — извлекаются из базы данных, затем сохраняются в кеш с TTL 60 секунд. Изменение данных приводит к актуализации или удалению соответствующих кешированных записей. В приложении для работы с кешем используется сервис UserCacheService, взаимодействующий с Redis через RedisTemplate. Каждому объекту присваивается уникальный ключ, формируемый с префиксом ( user:v1: ) и идентификатором пользователя.

**Асинхронная обработка событий**

Асинхронное взаимодействие между микросервисами реализовано с использованием брокера сообщений RabbitMQ. Коммуникация построена по классической модели producer/consumer, позволяющей разделить генерацию событий и их обработку во времени.

* **Producer (UserService);**  
  Внутри пользовательского сервиса при выполнении значимых действий (например, создание пользователя) формируется событие, которое публикуется в очередь сообщений с помощью компонента UserEventPublisher. Это позволяет зафиксировать факт операции без выполнения дополнительных действий в основном потоке обработки запроса.
* **Consumer (NotificationService).**Сервис уведомлений подписан на соответствующую очередь и непрерывно прослушивает поступающие события. При получении сообщения он выполняет отложенные действия: записывает лог, инициирует отправку уведомлений, взаимодействует с внешними системами.

**Репозиторий в Github:** [**https://github.com/denisovdev/architecture/**](https://github.com/denisovdev/architecture/)

# Тестирование

## Кеширование

A screenshot of a computer

Description automatically generated

Рисунок 1. Создание пользователя.

A screenshot of a computer

Description automatically generated

Рисунок 2. Ответ сервера.

A screenshot of a computer

Description automatically generated

Рисунок 3. Запрос на получение данных о пользователе.

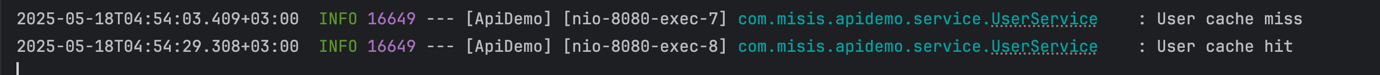


Рисунок 4. Логи API-сервиса.

Таким образом, при первом запросе, данные о пользователе были получены из базы данных, после чего помещены в кеш. При повторном запросе, данные были получены из кеша.

## RabbitMQ

A screenshot of a computer

Description automatically generated

Рисунок 5. Создание пользователя.

Полученные из запроса данные должны отправиться в очередь RabbitMQ сервисом UserService и получены сервисом NotificationService.



Рисунок 6. Логи UserService.

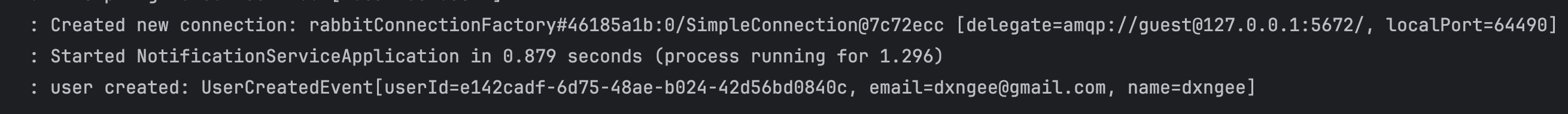


Рисунок 7. Логи NotificationService.

Убедившись, что данные дошли и новый пользователь был создан, можно сделать вывод, что асинхронное межсервисное взаимодействие реализовано корректно.

# Заключение

В рамках курсовой работы была реализована распределённая система на основе микросервисной архитектуры, ориентированная на устойчивость, масштабируемость и соответствие современным требованиям промышленной разработки. Проект строился с использованием ключевых технологий, обеспечивающих стабильную и эффективную работу всех компонентов приложения.

В качестве основы для развёртывания и управления окружением использовалась контейнеризация с помощью Docker. Все сервисы были упакованы в отдельные контейнеры, что позволило упростить процесс настройки, обеспечило предсказуемость среды и облегчило масштабирование при необходимости увеличения нагрузки. Для взаимодействия между контейнерами использовался Docker Compose, в котором описаны конфигурации необходимых инфраструктурных компонентов, таких как PostgreSQL, Redis и RabbitMQ.

Особое внимание было уделено производительности системы. Для ускорения отклика и снижения количества обращений к основной базе данных было реализовано кеширование с использованием Redis (KeyDB) по стратегии cache-aside. Это позволило хранить в памяти наиболее востребованные данные, тем самым значительно снизив задержки при повторных запросах и уменьшив нагрузку на хранилище.

Для обеспечения асинхронной обработки задач и минимизации времени выполнения основного бизнес-потока была реализована модель взаимодействия по принципу producer–consumer. Связь между микросервисами осуществлялась через очередь сообщений RabbitMQ, что позволило организовать надёжную доставку сообщений, обрабатывать фоновые задачи отдельно от основного приложения и добиться высокой отказоустойчивости.

Таким образом, в ходе выполнения курсового проекта была разработана полноценная распределённая система с применением современных инженерных подходов. Полученное решение демонстрирует принципы построения надёжной, расширяемой и обслуживаемой архитектуры, аналогичной тем, что используются в реальных проектах промышленного уровня.

# Список литературы

1. Java Documentation — <https://docs.oracle.com/en/java/>
2. Spring Documentation — <https://docs.spring.io/spring-boot/index.html>
3. Docker Documentation — <https://docs.docker.com>
4. Redis Documentation — <https://redis.io/docs>
5. RabbitMQ Documentation — <https://www.rabbitmq.com/docs>
6. PostgreSQL Documentation — <https://www.postgresql.org/docs/>