МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

# РАЗРАБОТКА И РЕАЛИЗАЦИЯ МЕХАНИЗМА РАБОТЫ С ЗАКАЗАМИ ИНТЕРНЕТ-МАГАЗИНА С ПРИМЕНЕНИЕМ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 411 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Буйкевича Ярослава Георгиевича

Научный руководитель
доцент, к. т. н. _____ В. М. Соловьёв

Заведующий кафедрой
доцент, к. ф.-м. н. _____ С. В. Миронов

Саратов 2025

# ВВЕДЕНИЕ

Актуальность разработки распределённых микросервисных приложений обусловлена развитием облачных технологий и необходимостью горизонтального масштабирования современных онлайн-сервисов. В частности, рынок онлайн-доставки продуктов питания демонстрирует высокие темпы роста: пользователи всё чаще выбирают удобные и оперативные решения для заказа товаров первой необходимости и свежих продуктов.

Монолитные приложения, реализующие полный цикл работы сервиса доставки продуктов, испытывают ряд ограничений: низкая отказоустойчивость, сложности масштабирования отдельных функциональных блоков, увеличенное время развертывания и обновления, а также трудности сопровождения. Микросервисная архитектура позволяет разделить систему на независимые сервисы, отвечающие за конкретные бизнес-задачи, что повышает гибкость разработки, упрощает масштабирование и ускоряет выпуск новых возможностей.

В рамках данного дипломного проекта реализован прототип аналога сервиса «Самокат» — платформы быстрой доставки продуктов питания. Проект охватывает ключевые этапы работы сервиса: формирование заказа пользователем и его дальнейшая обработка системой. Физическая доставка курьером не рассматривается, чтобы сосредоточиться на архитектурных и интеграционных аспектах микросервисного решения.

**Цель бакалаврской работы:** разработать и реализовать три микросервиса, отвечающих за формирование заказа, обработку заказа и симуляцию платёжной системы, которые будут демонстрировать основные принципы взаимодействия компонентов в распределённой архитектуре для сервиса доставки продуктов.

Поставленная цель определила **следующие задачи**:

1. **Анализ предметной области:** исследовать бизнес-процессы онлайн-доставки продуктов (выбор товаров, корзина, оформление заказа, оплата).
2. **Проектирование архитектуры:** спроектировать высокоуровневую архитектуру микросервисов, включающую сервис формирования заказа, сервис обработки заказа, брокер сообщений, базы данных и API-шлюз.
3. **Реализация микросервиса формирования заказа:** обеспечить приём запросов от клиентов, валидацию данных, хранение информации о новом заказе и публикацию событий в шину сообщений.

4. **Реализация микросервиса обработки заказа:** потреблять события о новых заказах, выполнять логику подтверждения, проведения оплаты и изменения статусов.
5. **Реализация микросервиса оплаты заказа:** данный сервис должен уподобляться платёжной системе, выдавать запрашивающей стороне статус оплаты
6. **Организация взаимодействия:** настроить обмен сообщениями между микросервисами через брокер (RabbitMQ/Kafka), выбрать и сконфигурировать хранилище данных (реляционная или NoSQL БД).
7. **Логирование и обработка ошибок:** сделать сервис отказоустойчивым и настроить подробное логирование для технической поддержки, чтобы легко устранять неполадки в ходе работы приложения
8. **Тестирование:** провести ручное тестирование основных сценариев.
9. **Документирование OpenAPI:** разработать документацию в Swagger по работе с API для удобства внедрения микросервисов.

Результатом стал демонстрационный прототип, отражающий ключевые аспекты микросервисного подхода в контексте быстрого сервиса доставки продуктов «Самокат». Дальнейшее развитие проекта может включать модули управления складом, систему мониторинга доставки курьеров и оптимизацию маршрутов с учётом загруженности.

**Структура и объём работы**. Бакалаврская работа состоит из введения, 5 разделов, заключения, списка использованных источников и 7 приложений. Общий объём — 92 страницы, из них 53 страницы — основное содержание, включая 11 рисунков, Flash-носитель в качестве приложения, список использованных источников информации — 22 наименования.

**Краткое содержание работы**

**Первый раздел «Понятие ипринципы микросервисной архитектуры»** посвящён анализу микросервисной архитектуры как современного подхода к проектированию программных систем. В нём рассматриваются ключевые отличия микросервисной архитектуры от традиционной монолитной модели, где приложение представляет собой единый блок кода с тесной связью компонентов. Подчёркивается, что микросервисная архитектура разбивает приложение на независимые сервисы, взаимодействующие через API, что обеспечивает модульность, распределённую разработку и гибкость.

**Основные преимущества микросервисной архитектуры включают:**

— **Масштабируемость**: возможность горизонтального масштабирования отдельных сервисов, гранулярное распределение ресурсов и автоматическое управление нагрузкой с помощью оркестраторов (например, Kubernetes).

— **Независимость компонентов**: автономная разработка и деплой сервисов, свобода выбора технологий (языков программирования, баз данных) для каждого сервиса, изоляция жизненных циклов.

— **Отказоустойчивость**: изоляция сбоев, применение шаблонов проектирования (Circuit Breaker, Bulkhead) и распределённые системы мониторинга для минимизации последствий ошибок.

Особое внимание уделено принципам проектирования микросервисов:

— Чёткое определение границ ответственности сервисов, изоляция данных и строгость интерфейсов.

— Независимость жизненного цикла: параллельная разработка, тестирование и деплой, поддержка обратной совместимости API.

— Надёжность взаимодействия: версионирование API, документирование контрактов (OpenAPI), выбор протоколов (HTTP/REST, gRPC, очереди сообщений).

— Система мониторинга и управления отказами: централизованный сбор метрик, автоматическое восстановление и изоляция проблемных сервисов.

Раздел демонстрирует, что микросервисная архитектура обеспечивает масштабируемость, гибкость и отказоустойчивость, что делает её предпочтительной для сложных, быстро развивающихся систем.

**Второй раздел «Анализ существующих подходов и технологий реали-**

**зации микросервисов»** посвящён анализу современных подходов и технологий реализации микросервисных архитектур. В работе исследуются ключевые инструменты контейнеризации (Docker) и оркестрации (Kubernetes), обеспечивающие стандартизацию развёртывания, масштабируемость и изоляцию сервисов. Рассмотрены два основных способа интеграции микросервисов:

- **Синхронная интеграция** через RESTful API, HTTP-запросы и GraphQL, характеризующаяся низкой задержкой и простотой отладки, но требующая обработки таймаутов и каскадных сбоев;
- **Асинхронная интеграция** с использованием очередей (Apache Kafka, RabbitMQ), позволяющая повысить отказоустойчивость и разделить критичные по времени операции от фоновых процессов.

Проведён анализ успешного внедрения микросервисов в крупных компаниях:

- **Netflix** — переход с монолита на десятки автономных сервисов, обеспечивающих высокую доступность и гибкость;
- **Amazon** — распределение бизнес-процессов по микросервисам с собственными базами данных и асинхронной обработкой событий;
- **Netflix** — использование контейнеризации и брокеров сообщений для персонализации рекомендаций и масштабирования инфраструктуры.

Обобщены принципы эффективного внедрения микросервисов: автономность команд, выбор адекватных механизмов хранения и обмена данными, а также стратегии управления отказами. Результаты анализа подтверждают возможность масштабирования, повышения отказоустойчивости и сокращения времени вывода изменений в производственную среду.

**Третий раздел «Постановка задачи и требования к разрабатываемой системе»** постановке задачи и разработке требований к системе управления заказами в сфере электронной коммерции. В работе рассмотрена архитектура сервиса доставки продуктов «Самокат», включающего ключевые компоненты: клиентский интерфейс, сервисы формирования и обработки заказа, навигации курьеров, платёжную систему. Описаны функциональные взаимодействия между участниками: проверка наличия товаров, расчёт стоимости, подтверждение оплаты, отслеживание доставки.

Для системы определены функциональные требования, включающие регистрацию пользователей, управление корзиной, выбор способов доставки и

оплаты, валидацию данных, отслеживание статуса заказов, историю покупок и возможность их отмены. Особое внимание уделено асинхронному отказоустойчивому взаимодействию сервисов с гарантией доставки данных, а также разделению информации на временные (корзина, сессии) и постоянные данные (история заказов, продукты).

В рамках проекта реализованы сервисы формирования и обработки заказа, в то время как функционал навигации курьеров, отслеживания их геолокации и связи с клиентами остаётся за рамками из-за ограничений времени и масштаба разработки. Отмечено, что полноценная реализация системы требует участия команды специалистов различных профилей, как это практикуется в крупных аналогах на рынке.

**Четвёртый раздел «Постановка задачи и требования к разрабатываемой системе»** посвящён проектированию архитектуры приложения, реализованного в стиле микросервисов. Разработанная система включает три независимых микросервиса:

1. **CatalogService** — REST API для формирования заказа, поиска товаров и управления корзиной с использованием Swagger для документации.
2. **OrderHandlerService** — обрабатывает заказы, проверяет наличие товаров, формирует запросы на доставку и инициирует оплату через внешний сервис.
3. **PaymentService** — симулирует платёжную систему, генерирует ссылки для оплаты и передаёт статусы через REST API (без Kafka, так как реальные платёжные системы обычно внешние).

Кроме того, в данном разделе описан выбранный способ связи между микросервисами — Apache Kafka. При помощи Docker Compose был поднят кластер, содержащий все необходимые хранилища данных — Redis, MongoDB, PostgreSQL. В конце раздела описаны вообще все технологии, использованные в работе.

**Пятый раздел «Проектирование, разработка и тестирование сервисов»** посвящён проектированию и разработке всех трёх сервисов. Он разбит на три подраздела.

Первый подраздел посвящён проектированию, разработке и тестированию CatalogService. Сервис должен предоставлять каталог товаров в виде древовидной структуры с возможностью поиска. Основные сущности: `Product` и

Category.

Сущность Product представляет товар и содержит поля: уникальный идентификатор с автоинкрементом, обязательное имя, описание в формате длинного текста, стоимость, ссылку на изображение и связь с категорией через внешний ключ. Для работы с этой сущностью используются аннотации JPA, такие как @Entity (определяет класс как сущность), @Data (автоматически генерирует геттеры и сеттеры), @Column (задаёт параметры полей базы данных) и @ManyToOne (указывает связь с категорией).

Сущность Category описывает категорию товаров. Она включает идентификатор, обязательное имя, ссылку на изображение, ссылку на родительскую категорию и список дочерних категорий. Вложенность категорий реализуется через рекурсивные ссылки, например, категория "мясо"может содержать подкатегории "свинина "курица"и т.д.

Параметры отношений между категориями обеспечивают гибкость управления данными: каскадные операции (cascade = CascadeType.ALL) применяются ко всем дочерним элементам, удаление "сиротских"записей (orphanRemoval = true) происходит автоматически, а дочерние категории загружаются немедленно (fetch = FetchType.EAGER) при запросе родительской категории.

Для работы с сущностями Product и Category создаются репозитории, реализующие интерфейс JpaRepository. В репозитории ProductRepository определяется сложный нативный запрос для получения продуктов из выбранной категории и всех её дочерних. Это позволяет, например, отображать товары категории «мясо» вместе с подкатегориями «курица», «говядина» и т.д. Пагинация (Pageable) используется для эффективного отображения больших списков товаров.

В репозитории CategoryRepository реализуется операция получения корневых категорий (без родителей), что необходимо для формирования структуры каталога в API.

Сервисный слой реализуется через класс с аннотацией @Service, где зависимости внедряются через конструктор.

Класс ProductController реализует REST-методы для работы с каталогом товаров. Он использует аннотации, такие как @Tag для группировки методов в документации Swagger UI, @CrossOrigin для разрешения кросс-доменных запросов, @RestController для автоматической сериализации возвращаемых

значений в JSON и `@RequestMapping("/api/v1")` для задания базового пути всех методов.

Метод `getProductsByCategory` обрабатывает GET-запрос по пути `/categories/` где `categoryId` определяется из URL. Параметры пагинации — `pageNumber` и `pageSize` — задаются через запрос с дефолтными значениями 1 и 10. Метод возвращает список DTO-продуктов текущей категории и её дочерних, используя сервисный слой.

В методе применяется логирование (`log.info`), чтобы фиксировать действия пользователя и упрощать диагностику ошибок, таких как 5xx-статусы. Остальные методы контроллера реализованы аналогично, их код приведён в приложении.

Система авторизации реализована через номер телефона пользователя. Сессии хранятся в Redis, где данные управляются через уникальный UUID. Конфигурация Redis в Spring осуществляется через класс `RedisConfiguration`, где создаются бины `RedisConnectionFactory` и `RedisTemplate`. Первый задаёт подключение к локальному инстансу Redis, второй определяет сериализацию ключей (в виде строк) и значений (в формат JSON), что позволяет работать с объектами.

Для управления сессиями разработан класс `SessionRepository`, использующий `RedisTemplate`. Он хранит идентификатор сессии, данные пользователя, корзину, адрес и способ оплаты. Методы `getSession`, `putSession` и `hasKey` обеспечивают взаимодействие с Redis. Приватные методы оборачиваются в публичные методы с аннотацией `@Retryable`, которая позволяет задавать повторные попытки и задержки при сбоях. Это достигается через паттерн *Proxy*, где обёртки методов добавляют логику повторных вызовов и логгирования, например, при операции сохранения сессии:

```
@Retryable(maxAttempts = 3, backoff = @Backoff(delay = 1000, multiplier = 2))
public void save(Session session) {
putSession(session);
}
```

Использование Redis обеспечивает быстрое хранение и доступ к сессиям, а аннотация `@Retryable` повышает надёжность работы с кэшем.

Корзина реализована как класс `Cart`, использующий хэш-таблицу для хранения продуктов, где ключ — ID товара, а значение — количество. При со-

здании сессии генерируется UUID, который сохраняется в Redis и возвращается в заголовке запроса для последующих операций.

Данные пользователя хранятся в PostgreSQL (телефон и имя), а информация об адресах, способах оплаты и заказах — в MongoDB. Это связано с менее структурированной природой данных. Для работы с MongoDB классы помечаются аннотацией `@Document`, поля с индексами — `@Indexed`. Пример: сущность оплаты содержит аннотированные поля для поиска по ID пользователя.

Авторизация пользователя включает проверку валидности телефона и существования сессии. Если пользователь не найден, формируется временный объект, который сохраняется в MongoDB только после совершения заказа. Сессия с привязанным пользователем обновляется в Redis. Реализация репозиториев и классов-сущностей описана в приложениях.

Класс `SessionService` содержит методы для работы с адресами и способами оплаты в рамках сессии. Они не создают новые записи в MongoDB, а управляют ссылками на уже существующие данные. Пользователь может добавлять или изменять адреса и способы оплаты на любом этапе формирования заказа — эти изменения сохраняются в сессии до момента завершения заказа.

На уровне контроллера реализованы операции создания сессии и авторизации. Метод `createSession` генерирует уникальный токен, который возвращается клиенту для последующего использования. Метод `authorization` связывает пользователя с сессией через заголовок `Authorization` и данные из тела запроса (`UserDto`), преобразуемого из JSON автоматически.

Для обработки POST-запросов используется аннотация `@PostMapping`, а `@SecurityRequirement` позволяет задавать API-ключ в Swagger для тестирования аутентифицированных запросов. Затем идёт демонстрация Swagger в картинках.

Для управления корзиной и формированием заказов реализованы классы `CartController` и `CurrentOrderController`. Метод `addToCart()` добавляет продукт в корзину по ID товара и ключу сессии, увеличивая количество дубликатов, если товар уже существует. Метод `deleteFromCart()` удаляет товар или уменьшает его количество. Все изменения сохраняются в Redis через хэш-таблицу `Map<Long, CartItem>`, где ключ — ID продукта, а значение — объект с количеством.

Класс `CurrentOrderController` отвечает за завершение заказа. После

выбора адреса и способа оплаты данные сохраняются в MongoDB, а событие отправляется в Kafka. Используется Apache Kafka для асинхронной передачи сообщений между микросервисами:

— `CatalogService` выступает продюсером топиков `newOrder` (новый заказ) и `newStatusClient` (обновление статуса клиента).

— Для топика `newStatusHandler` он работает как консьюмер, получая ответы от сервиса обработки заказов.

При формировании заказа `CatalogService` отправляет запрос к `PaymentService` с указанием Webhook-URL. Платёжная система использует этот URL для отправки статуса оплаты. Затем показаны скриншоты примеров взаимодействия:

— Сформированная корзина: содержит список товаров и их количество.

— Адрес доставки и способ оплаты: сохраняются в MongoDB.

— Подтверждённый заказ: включает данные пользователя, выбранные параметры и статус.

Конфигурация Kafka в Spring аналогична Redis, но вместо словарей используются топики — логические очереди для передачи сообщений между продюсерами и консьюмерами. Эта архитектура позволяет масштабировать систему и обеспечивает надёжную интеграцию микросервисов.

В следующем подразделе описывается разработка и тестирование `Order HandlerService` . Сервис выполняет следующие ключевые функции:

1. **Проверка наличия товаров на складе** — для каждого товара в заказе сравнивается запрошенное количество с актуальным остатком. При недостатке товара заказ отменяется, а уведомление отправляется в очередь для первого сервиса.
2. **Интеграция с Kafka** — сервис слушает топики newOrder и newStatus, чтобы получать сообщения о новых заказах и изменениях статусов. Пример реализации: метод newOrderListener с аннотацией *@KafkaListener* обрабатывает сообщения из топика newOrder.
3. **Обработка вебхуков от платежной системы** — закрытое API сервиса принимает уведомления об оплате/отмене заказа. При успешной оплате статус заказа обновляется на PAID , а информация передаётся первому сервису.
4. **Взаимодействие с компонентами системы** — при отмене заказа сервис отправляет уведомление в очередь, что позволяет первому сервису уда-

лить заказ из кэша.

Разработанный сервис обеспечивает надёжную обработку заказов, синхронизацию данных между компонентами системы и своевременное реагирование на изменения статусов.

Третий подраздел посвящён разработке и тестированию PaymentService — симуляции платёжной системы, абстрагирующей процесс интернет-оплаты. Сервис реализует два сценария: успешную и неуспешную зарезервированную оплату. Логика взаимодействия включает инициализацию транзакции через передачу данных о способе и сумме оплаты, сохранение информации в Redis, генерацию уникального кода транзакции и возврат его клиенту. Для завершения оплаты используется Webhook-механизм: по полученному коду отправляется уведомление о результате на указанный URL. Решение демонстрирует ключевые этапы обработки платежей в распределённых системах.

# ЗАКЛЮЧЕНИЕ

В ходе дипломного проекта был разработан прототип микросервисного решения для сервиса быстрого заказа и обработки заказа продуктов питания «Самокат». Создание отдельных сервисов формирования заказа и обработки заказа продемонстрировало преимущества распределённой архитектуры: автономность компонентов, гибкость масштабирования и отказоустойчивость. Данные микросервисы легко интегрировать с микросервисами, которые будут строить оптимальный маршрут курьера по сбору и доставке заказа по адресу, указанному в заказе. Именно микросервисная архитектура даёт возможность всевозможных интеграций.

Основные результаты работы:

- Проведён анализ бизнес-процессов онлайн-доставки и определены ключевые этапы взаимодействия клиентов и системы.
- Спроектирована архитектура, включающая три микросервиса, брокер сообщений и систему хранения данных, что обеспечило чёткое разделение ответственности и простоту интеграции.
- Реализован микросервис формирования заказа с валидацией входящих запросов и публикацией событий в шину сообщений.
- Реализован микросервис обработки заказа с обработкой платежей, обновлением статусов и предоставлением REST API для внешних клиентов.
- Настроено взаимодействие сервисов через Kafka и выбрана оптимальная структура базы данных для хранения информации о заказах, применены различные технологии хранилищ данных, такие как PostgreSQL, MongoDB и Redis.
- Проведено ручное тестирование, подтвердившее корректность работы и устойчивость системы при различных сценариях.

Практическая ценность проекта заключается в демонстрации этапов разработки и интеграции микросервисов для онлайн-сервиса доставки, что может служить основой для дальнейшего расширения функциональности и внедрения в реальных продуктах.