СЕКЦИЯ 12 Информационные технологии. Электронно-вычислительные системы и комплексы

УДК 004.62

**УПРАВЛЕНИЕ ДАННЫМИ В МИКРОСЕРВИСНЫХ АРХИТЕКТУРАХ**

***Сафронов Никита Сергеевич***nikita\_safronov@outlook.com

КФ МГТУ им. Н.Э. Баумана

***Гагарин Юрий Евгеньевич*** gagarin\_ye@bmstu.ru

КФ МГТУ им. Баумана

В описываемом исследовании рассматриваются актуальные проблемы управления данными в микросервисных архитектурах, предлагая решения на основе современных подходов и паттернов. В ходе исследования обсуждаются основные вызовы, такие как согласованность данных, доступ к данным, их хранение и избыточность. В статье представлен обзор популярных механизмов управления данными, таких как базы данных на уровне сервиса, API для доступа к данным, событийно-ориентированное взаимодействие и паттерн CQRS. Предложенные решения позволяют эффективно управлять данными в распределенных системах, обеспечивая согласованность, масштабируемость и производительность приложений на основе микросервисной архитектуры.

***Ключевые слова:*** *микросервисная архитектура, управление данными, согласованность данных, доступ к данным, хранение данных, избыточность данных, базы данных на уровне сервиса, API для доступа к данным, событийно-ориентированное взаимодействие, CQRS.*

Микросервисные архитектуры становятся все более популярными для разработки программного обеспечения [3]. Этот подход предоставляет множество преимуществ, таких как увеличение гибкости, масштабируемости и скорости разработки, но одновременно создает сложности в управлении данными. Управление данными в микросервисных архитектурах имеет свои собственные вызовы [4], такие как:

*Согласованность данных:* в случае использования отдельных баз данных каждый из сервисов управляет своей собственной базой данных, отсутствие централизованной точки доступа к данным усложняет обеспечение их целостности, поскольку изменения в одном сервисе могут требовать соответствующих обновлений в других сервисах.

*Дублирование данных:* в архитектуре микросервисов каждый сервис может иметь свою собственную базу данных, что может привести к дублированию данных между несколькими сервисами; дублирование данных может привести к несогласованности, увеличению затрат на хранение и снижению производительности системы.

*Доступ к данным:* каждый сервис имеет свою базу данных, что может затруднить доступ к данным между сервисами.

*Хранение данных:* каждый сервис имеет свою базу данных, что может привести к увеличению затрат на хранение и снижению производительности системы.

Для решения этих проблем существуют различные механизмы и паттерны управления доступом к данным, которые выполняют свою задачу в совокупности. Первым из них является паттерн *отдельных баз данных.* Каждый микросервис имеет свою собственную отдельную базу данных, к которой он имеет эксклюзивный доступ. Каждый сервис управляет только своими данными, что обеспечивает независимость и изоляцию. Это позволяет сервисам быть более автономными и уменьшает вероятность конфликтов данных между сервисами. Кроме того, этот подход обеспечивает лучшую масштабируемость, так как каждая база данных может быть масштабирована независимо от других.

В *шаблоне композиции API* несколько микросервисов объединяются для предоставления единого API клиенту. Этот шаблон обеспечивает высокую гибкость в управлении данными, поскольку каждый микросервис может управлять своими данными независимо. В этом шаблоне используется единый API-шлюз для предоставления единообразного интерфейса для всех сервисов, позволяя клиентам получать доступ к данным из нескольких сервисов через единый API.

Одним из ключевых механизмов взаимодействия между сервисами является *событийно-ориентированная архитектура* (или EDA) [1]. Этот подход основан на обмене сообщениями, или событиями, между компонентами системы, что позволяет создать легкую, гибкую и масштабируемую архитектуру. В EDA события являются центральным механизмом передачи информации и управления, и каждый компонент реагирует на определенные события и генерирует свои собственные события.

Основные принципы событийно-ориентированной архитектуры:

*Локальная автономия*: каждый компонент системы независимо обрабатывает события и принимает решения на основе полученной информации, что обеспечивает локальную автономию и гибкость.

*Отправка и подписка*: события отправляются и распространяются по системе через механизмы отправки и подписки, а компоненты могут подписываться на определенные типы событий и реагировать на них соответствующим образом.

*Асинхронность*: взаимодействие между компонентами происходит асинхронно, что позволяет повысить отзывчивость системы и обеспечить масштабируемость.

*Паттерн Saga* — это механизм для обеспечения согласованности данных в распределенных системах, особенно в микросервисной архитектуре. Этот паттерн позволяет управлять транзакциями, которые воздействуют на данные, распределенные между несколькими сервисами, и гарантировать их согласованность в случае возникновения ошибок. Этот паттерн предлагает способ обработки долгих и сложных транзакций без блокировки и ожидания ответа от всех участвующих сервисов. Вместо этого он использует набор небольших транзакций, известных как шаги, каждый из которых либо успешно завершается, либо отменяется.

Оркеструемые саги и хореографируемые саги — это два основных подхода к реализации паттерна Saga в распределенных системах.

*Оркеструемые саги* предполагают наличие центрального компонента, называемого оркестратором, который координирует и управляет выполнением всех шагов транзакции. Оркестратор определяет порядок выполнения шагов, запускает их и обрабатывает ошибки, при необходимости вызывая компенсирующие действия.

В *хореографируемых сагах* отсутствует централизованный оркестратор, и каждый сервис самостоятельно управляет своими действиями и реакциями на изменения состояния системы. Каждый сервис знает о своих собственных действиях и о том, как реагировать на сообщения и события от других сервисов.

Оба подхода имеют свои преимущества и недостатки, и выбор между ними зависит от конкретных требований и контекста системы.

В *паттерне CQRS* (Command Query Responsibility Segregation) один микросервис отвечает за обработку команд, в то время как другой микросервис отвечает за обработку запросов. Основная идея CQRS заключается в том, что операции записи данных (команды) и операции чтения данных (запросы) должны иметь разные модели данных и механизмы доступа. Вместо того, чтобы использовать одну унифицированную модель данных для всех операций, CQRS рекомендует разделить данные на две модели: одну для обработки команд и другую для обработки запросов.

Преимущества CQRS включают улучшенную производительность и масштабируемость за счет оптимизации моделей данных для конкретных операций, а также более гибкую архитектуру, позволяющую легко вносить изменения в систему. Однако внедрение CQRS может быть сложным и требует тщательного проектирования и реализации.

**Литература**

[1] Беллермар А. Создание событийно-управляемых микросервисов. Санкт-Петербург, БХВ-Петербург, 2022, 320 с.

[2] Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга. Санкт-Петербург, Питер, 2019, 544 с.

[3] Кочер П.С. Микросервисы и контейнеры Docker. Москва, ДМК Пресс, 2018, 242 с.

[4] Ньюман С. От монолита к микросервисам. Санкт-Петербург, БХВ-Петербург, 2021, 266 с.

**DATA MANAGEMENT IN MICROSERVICES ARCHITECTURES**

**Safronov Nikita Sergeevich** nikita\_safronov@outlook.com

BMSTU (KB)

**Gagarin Yuri Evgenievich** gagarin\_ye@bmstu.ru

BMSTU (KB)

*The described research examines the current challenges of data management in microservices architectures, offering solutions based on modern approaches and patterns. Throughout the study, key challenges such as data consistency, data access, storage, and redundancy are discussed. The article provides an overview of popular data management mechanisms such as database per service, API for data access, event-driven interaction, and the CQRS pattern. The proposed solutions enable effective data management in distributed systems, ensuring consistency, scalability, and performance of applications based on microservices architecture.*

***Keywords****: microservices architecture, data management, data consistency, data access, data storage, data redundancy, database per service, API for data access, event-driven interaction, CQRS.*