СТАТЬИ НА РУССКОМ ЯЗЫКЕ

ИНФОРМАТИКА, ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА И УПРАВЛЕНИЕ

DOI - 10.32743/UniTech.2024.128.11.18560

ОПТИМИЗАЦИЯ ВЫСОКОНАГРУЖЕННЫХ ВЕБ-ПРОЕКТОВ С ИСПОЛЬЗОВАНИЕМ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ

Шумилов Михаил Игоревич
технический директор компании Vadimages,
США, штат Вашингтон, Ванкувер,
Украина, г. Харьков
E-mail: michael@vadimages.com

OPTIMIZATION OF HIGH-LOAD WEB PROJECTS USING MICROSERVICE ARCHITECTURE

Mykhailo Shumilov
CTO at Vadimages,
USA, WA, Vancouver,
Ukraine, Kharkiv

АННОТАЦИЯ

В данной статье подробно исследуются методы оптимизации высоконагруженных веб-проектов с применением микросервисной архитектуры. Основная цель работы заключается в проведении всестороннего анализа эффективности использования микросервисного подхода для обеспечения гибкости, масштабируемости и высокой отказоустойчивости современных веб-приложений. В рамках исследования рассматриваются не только ключевые концепции микросервисной архитектуры, но и современные инструменты, такие как Docker и Kubernetes, которые играют важную роль в автоматизации и оркестрации процессов. Методология основана на анализе применения данных технологий для повышения общей производительности систем, улучшения управляемости и обеспечения легкой интеграции новых модулей в существующую инфраструктуру. Важным аспектом исследования является также изучение способов оптимизации времени отклика системы, повышения ее стабильности и устойчивости к сбоям. Результаты показывают, что внедрение микросервисной архитектуры не только сокращает время отклика и облегчает процесс интеграции новых модулей, но и существенно снижает эксплуатационные риски, одновременно повышая эффективность работы высоконагруженных систем.

В выводах подчеркивается, что использование данного подхода позволяет веб-приложениям быстро адаптироваться к изменяющимся требованиям бизнеса, обеспечивая при этом их стабильную и безопасную работу, что особенно важно в условиях постоянно растущих нагрузок и изменяющихся бизнес-потребностей.

ABSTRACT

This article examines in detail the methods of optimizing high-load web projects using microservice architecture. The main purpose of the work is to conduct a comprehensive analysis of the effectiveness of using the microservice approach to ensure flexibility, scalability and high fault tolerance of modern web applications. The research examines not only the key concepts of microservice architecture, but also modern tools such as Docker and Kubernetes, which play an important role in automation and orchestration of processes.

The methodology is based on the analysis of the application of these technologies to improve the overall performance of systems, improve manageability and ensure easy integration of new modules into existing infrastructure. An important aspect of the study is also the study of ways to optimize the response time of the system, increase its stability and resilience to failures. The results show that the introduction of a microservice architecture not only reduces response time and facilitates the integration of new modules, but also significantly reduces operational risks, while increasing the efficiency of highly loaded systems. The conclusions emphasize that using this approach allows web applications to quickly adapt to changing business requirements, while ensuring their stable and secure operation, which is especially important in conditions of constantly increasing loads and changing business needs.

Ключевые слова: микросервисная архитектура, оптимизация, высоконагруженные системы, масштабируемость, отказоустойчивость, Docker, Kubernetes.

Keywords: microservice architecture, optimization, high-load systems, scalability, fault tolerance, Docker, Kubernetes.

Библиографическое описание: Шумилов М.И. ОПТИМИЗАЦИЯ ВЫСОКОНАГРУЖЕННЫХ ВЕБ-ПРОЕКТОВ С ИСПОЛЬЗОВАНИЕМ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ // Universum: технические науки : электрон. научн. журн. 2024. 11(128). URL: *https://7universum.com/ru/tech/archive/item/18560*

Введение

В условиях стремительного роста объёмов данных и увеличения количества пользователей веб-приложений разработчики сталкиваются с необходимостью поиска эффективных решений для оптимизации высоконагруженных систем. Традиционные монолитные архитектуры, применявшиеся ранее, часто не справляются с вызовами современных реалий, такими как высокая нагрузка, сложность интеграции новых модулей и обновлений, а также трудности в поддержании отказоустойчивости.

В связи с этим, микросервисная архитектура становится всё более востребованной, так как она позволяет гибко адаптировать систему к изменяющимся требованиям и обеспечивать её стабильную работу даже при пиковых нагрузках.

Микросервисы представляют собой модульный подход к разработке программного обеспечения, при котором приложение разбивается на небольшие, независимые друг от друга сервисы. Каждый микросервис выполняет определённую функцию и взаимодействует с другими через чётко определённые API. Такой подход позволяет разработчикам легко изменять или обновлять отдельные компоненты системы, не затрагивая её целостность. Использование микросервисной архитектуры также предоставляет возможности для автоматизации процессов разработки и развертывания, что особенно актуально в контексте применения DevOps и CI/CD практик.

Актуальность темы обусловлена необходимостью создания высокопроизводительных и надёжных веб-приложений, способных справляться с большими объёмами данных и множеством параллельных запросов. Современные технологии, такие как Docker и Kubernetes, обеспечивают контейнеризацию и оркестрацию микросервисов, упрощая их развёртывание и управление. Благодаря этому разработчики могут быстро масштабировать сервисы в зависимости от текущей нагрузки, минимизируя риски возникновения ошибок и сбоев.

Цель работы заключается в проведении анализа эффективности микросервисного подхода в обеспечении гибкости, масштабируемости и отказоустойчивости веб-приложений.

Материалы и методы

Исторически термин «архитектура» связывался в первую очередь с возведением зданий и сооружений. В сфере программного обеспечения архитектура представляет собой структурный подход к созданию решений, способных удовлетворять бизнес-задачи, обеспечивать удобство использования и адаптацию к изменениям и развитию [1].

Микросервисная архитектура или просто микросервисы – это особый метод разработки программных систем, который пытается сосредоточиться на создании однофункциональных модулей с четко определенными интерфейсами и операциями. Ниже на рисунке 1 будет отражен принцип работы микросервисной архитектуры.

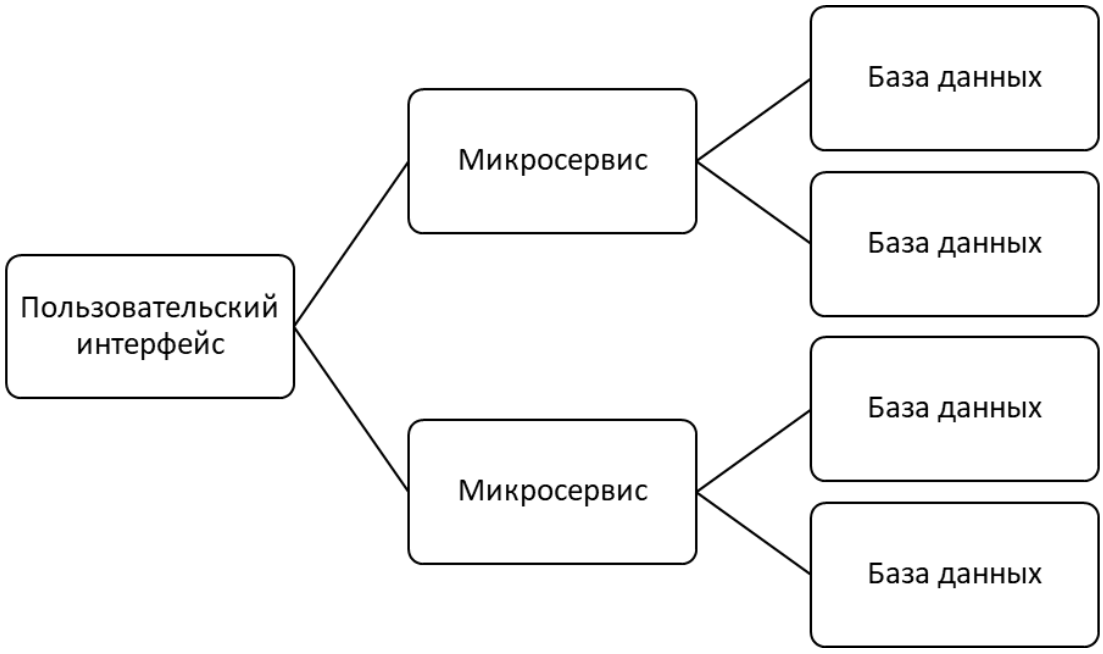

Рисунок 1. Принцип работы микросервисной архитектуры [2]

Микросервисная архитектура предоставляет множество возможностей для Agile- и DevOps-команд, что подтверждается практикой таких крупных технологических компаний, как Netflix, eBay, Amazon, Twitter и PayPal, которые постепенно отказались от монолитных структур в пользу микросервисов. Микросервисная архитектура предлагает решение этих проблем за счёт максимального разделения на модули. Каждый микросервис представляет собой небольшую независимую службу, которая функционирует в рамках собственного процесса и может быть развёрнута автономно, без воздействия на остальные компоненты.

Такие службы допускают использование различных языков программирования и подходов к организации баз данных, что позволяет адаптировать их к специфическим требованиям. В то же время, это повышает гибкость и масштабируемость системы, но требует динамического управления и интеграции. Чаще всего микросервисы взаимодействуют посредством API и опираются на инфраструктуру, развившуюся в среде RESTful и веб-сервисов, что упрощает тестирование их взаимодействия и интеграции при развёртывании [2].

Далее рассмотрим принципы архитектуры микросервисов.

1. Автономные и независимые сервисы. Каждый микросервис функционирует автономно, имея все необходимые ресурсы, что минимизирует взаимозависимости и поддерживает децентрализацию. Пример: отдельные сервисы для управления каталогом товаров и процессинга платежей..

2. API-агрегация. API обеспечивает совместимость микросервисов, независимо от технологий, и гибкость при интеграции новых сервисов.

3. Гибкость Микросервисы легко адаптируются к изменениям, позволяя модифицировать или удалять модули без влияния на другие части системы.

4. Масштабируемость. Архитектура поддерживает изменение ресурсов в зависимости от нагрузки, например, при увеличении трафика в пиковые периоды [3].

5. Непрерывный мониторинг. Постоянный мониторинг и системы журналов, метрик и трассировки помогают своевременно обнаруживать и устранять сбои.

6. Отказоустойчивость. Изоляция компонентов и резервирование позволяют минимизировать влияние сбоев на систему.

7. Балансировка нагрузки в реальном времени. Эти системы распределяют ресурсы между микросервисами в реальном времени, обеспечивая оптимальное использование процессорных мощностей и минимизируя время отклика на запросы.

8. Интеграция DevOps. Принципы DevOps автоматизируют разработку, тестирование и развертывание микросервисов с использованием Docker и Kubernetes.

9. Версионирование. Управление версиями позволяет обновлять сервисы, минимизируя влияние на пользователей и обеспечивая совместимость.

10. Высокая доступность. Микросервисы должны обеспечивать круглосуточную работу с минимальными простоями [4].

Для разработки и создания микросервисов существует обширный выбор инструментов, которые способны упростить и ускорить данный процесс. В ниже приведенном обзоре рассматриваются наиболее распространенные из них:

1. Spring Boot — это фреймворк, работающий на основе языка программирования Java. Он предоставляет широкий набор возможностей, значительно упрощающих создание и внедрение микросервисных архитектурных решений.

2. Node.js — платформа для разработки серверных решений на JavaScript, которая обеспечивает создание масштабируемых и производительных микросервисов, используя возможности данного языка.

Docker — это платформа с открытым исходным кодом, предназначенная для автоматизации процессов развёртывания и управления приложениями в контейнерах. Данный инструмент позволяет упаковывать микросервисы и развертывать их в изолированных контейнерных средах [5].

3. Kubernetes (или K8s) — система оркестрации контейнеров, обеспечивающая развертывание, масштабирование и управление микросервисными приложениями. Она предлагает обширный набор инструментов для управления сетевыми службами, хранения данных и автоматизации масштабирования компонентов микросервисов.

4. Apache Kafka — распределённая система для обработки потоковых данных, позволяющая интегрировать микросервисы, опираясь на событийную архитектуру. Она функционирует как посредник для обмена сообщениями между различными сервисами, обеспечивая их взаимодействие и синхронизацию [6].

Для реализации масштабируемости и отказоустойчивости применяются методы, такие как контейнеризация и балансировка нагрузки. Контейнеризация с использованием инструментов, например, Docker в связке с Kubernetes или OpenShift, позволяет быстро развертывать и управлять экземплярами микросервисов. Балансировка нагрузки, в свою очередь, обеспечивает равномерное распределение запросов между компонентами, что способствует стабильной работе системы [7].

Отказоустойчивость в процессе проектирования современных систем направлена на обеспечение их функциональности и работоспособности, несмотря на возможные неисправности. По мере роста масштабов и сложности инфраструктур вероятность возникновения сбоев возрастает. Чтобы поддерживать надёжность и непрерывность взаимодействия с пользователями, важно предусматривать меры отказоустойчивости. Эти меры включают такие методики, как резервирование данных, выявление и устранение ошибок, а также автоматическое восстановление системы для минимизации влияния проблем с оборудованием, программным обеспечением или сетью. Использование таких подходов позволяет обеспечить бесперебойную работу сервисов [8]. Далее в таблице 1 будет проведено сравнение отказоустойчивости от высокая доступность балансировки нагрузки.

Таблица 1.

Сравнение отказоустойчивости от высокая доступность балансировки нагрузки [9]

Аспект	Отказоустойчивость	Высокая доступность балансировки нагрузки
Определение	Гарантирует, что система продолжает работать должным образом, даже если некоторые компоненты выходят из строя.	Распределяет рабочие нагрузки между несколькими серверами, чтобы гарантировать, что ни один сервер не станет узким местом, обеспечивая доступность системы.
Основная цель	Поддерживайте функциональность системы, несмотря на сбои.	Максимально увеличьте время безотказной работы и использование ресурсов за счет балансировки нагрузки.
Ключевые методы	Резервирование, репликация, механизмы отработки отказа, обнаружение и исправление ошибок.	Алгоритмы распределения нагрузки (циклический перебор, наименьшее количество подключений и т.д.), проверки работоспособности, отказоустойчивость.
Избыточность	Высокий уровень резервирования (несколько компонентов выполняют одну и ту же задачу).	Умеренное резервирование (достаточное для балансировки нагрузки и обеспечения доступности).
Примеры	RAID (резервный массив независимых дисков), распределенные базы данных с репликацией.	Балансировка нагрузки DNS, балансировщики нагрузки приложений (например, NGINX, HAProxy).
Влияние на производительность	Может незначительно повлиять на производительность из-за проверок избыточности и обработки ошибок.	В целом производительность повышается за счет равномерного распределения рабочей нагрузки.

Таким образом, использование микросервисной архитектуры, подкрепленное современными технологиями и методами разработки, позволяет создавать масштабируемые, гибкие и устойчивые системы, удовлетворяющие требованиям бизнеса и пользователей.

Результаты и обсуждение

Планирование нагрузки создаваемых систем на длительный период времени зачастую является сложной задачей, так как прогнозирование конкретных показателей через полгода или год невозможно. Отсутствие заложенной возможности масштабирования может привести к снижению эффективности системы при увеличении нагрузки. В зависимости от особенностей проекта реализуется либо вертикальное, либо горизонтальное масштабирование. Вертикальное масштабирование подразумевает увеличение мощности отдельных элементов системы путем добавления ресурсов в рамках одного узла. Горизонтальное же масштабирование основывается на увеличении количества узлов, позволяя распределять нагрузку между ними.

Проще всего масштабировать сервисы и компоненты, если они следуют принципу stateless — не сохраняют состояния между запросами. В этом случае можно легко развернуть несколько экземпляров компонента или сервиса, распределяя нагрузку между ними равномерно. При использовании stateful-парадигмы необходимо учитывать наличие и сохранение состояния компонентов при балансировке.

Тестирование является неотъемлемой частью процесса запуска систем с высокой нагрузкой. Нагрузочные тесты используются для оценки соответствия системы определенным критериям эффективности (KPI). Из практики, увеличение времени обработки запросов часто связано с неэффективностью алгоритмов.

Применение кэширования данных способствует увеличению производительности при многократных однотипных запросах.

Проведение тестов помогает выявить узкие места, которые могли быть упущены на этапе проектирования.

Например, веб-сервис одного из проектов при нагрузочном тестировании показал низкую производительность, несмотря на свою архитектурную простоту. Анализ показал, что значительное количество времени занимают процессы сериализации и десериализации данных, что приводило к конвертации структур в двоичный формат и обратно. Решение было найдено быстро — благодаря редким изменениям данных удалось применить предсериализацию, что значительно сократило время обработки запросов. Ниже на рисунке 2 будет отражена оптимизация сериализации для внешнего API.

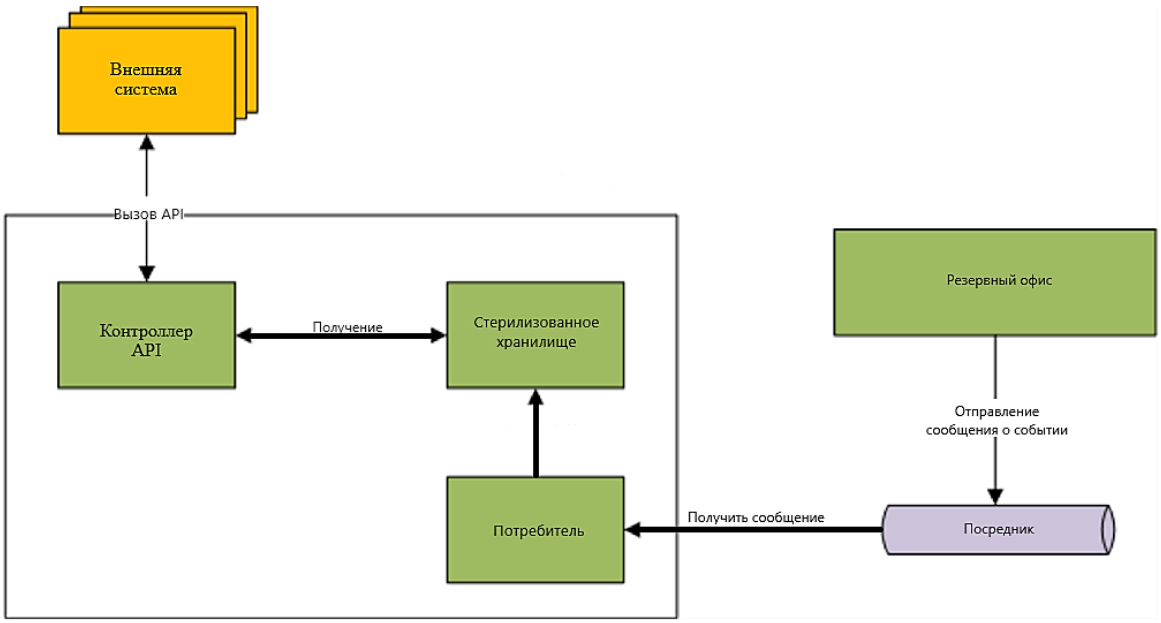

Рисунок 2. Оптимизация сериализации для внешнего API (предсериализация) [10]

Вместе с нагрузочными тестами, направленными на проверку соответствия системы установленным требованиям, проводятся также стресс-тесты. Их основная задача — определить максимальную нагрузку, которую может выдержать система без потери стабильности.

Метрики выступают в роли сенсоров, которыми оборудована каждая сложная система, особенно та, которая работает под высокой нагрузкой. Отсутствие таких показателей превращает программу в «черный ящик» — она имеет входные и выходные данные, но внутренние процессы остаются неясными.

Поэтому на этапе проектирования в архитектуру системы интегрируются специальные измерительные механизмы, которые обеспечивают подачу данных для мониторинговых систем.

В некоторых проектах для мониторинга активности и визуализации данных системы применялся стек ELK (ElasticSearch, Logstash и Kibana).

Этот инструмент обладает гибким интерфейсом, что позволяет быстро и удобно настроить нужные формы отчетности и визуализации (рис. 3.).

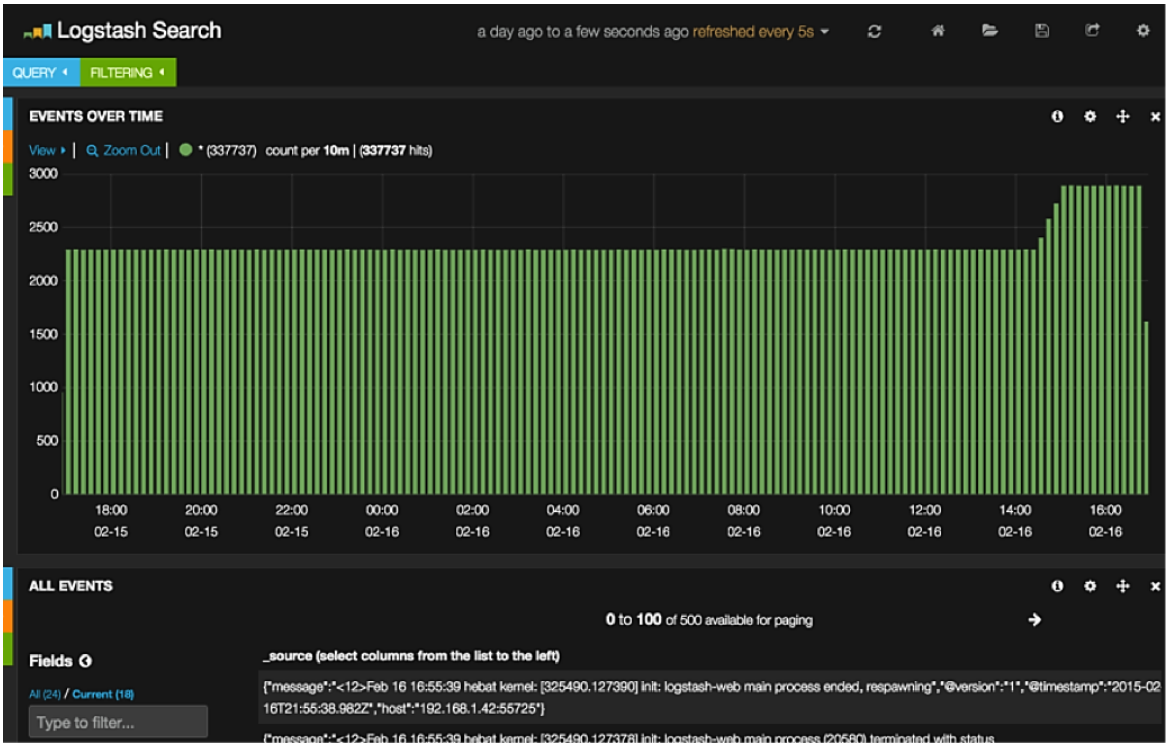

Рисунок 3. Мониторинг Kibana [10]

Основные преимущества микросервисной архитектуры при разработке высоконагруженных систем включают:

1. Возможность вносить изменения только в один модуль системы без необходимости полного развёртывания.

2. Размещение микросервисов, требующих частого масштабирования, на мощных серверах для повышения их эффективности.

3. Обеспечение отказоустойчивости системы: сбои в одном модуле не влияют на остальные.

4. Использование наиболее подходящих фреймворков для каждой конкретной разработки.

5. Замена устаревших микросервисов без негативного воздействия на другие части системы [11].

В свою очередь для решения проблем, связанных с микросервисной архитектурой, может быть применен событийно-ориентированный подход, включающий Event Sourcing и Command Query Responsibility Segregation (CQRS). Событийно-ориентированный подход представляет собой архитектурную парадигму, ориентированную на генерацию, обнаружение и обработку событий, а также на реакцию на них.

Основу веб-приложения в такой системе составляют события, обеспечивающие обмен данными между различными сервисами. Каждый сервис реагирует на конкретные события, при этом инициатор события не обладает информацией о том, какие сервисы его обрабатывают. Такой механизм способствует низкой связанности между микросервисами, что повышает согласованность данных в приложении и упрощает его масштабируемость.

Event Sourcing предполагает сохранение последовательности изменений, происходящих с состоянием веб-приложения.

Такая последовательность может использоваться как указатель на текущее состояние приложения либо как список зарегистрированных событий.

Применение этого подхода позволяет достичь высокой степени децентрализации операций чтения и модификации данных.

В контексте Event Sourcing агрегат выступает индикатором актуального состояния веб-приложения. Агрегат представляет собой группу элементов предметной области, имеющих тесные взаимосвязи (см. рис. 6). Для получения актуального состояния агрегата необходимо загрузить и воспроизвести соответствующие события.

Любое изменение агрегата требует создания нового события, которое описывает логику модификации его части, при этом сохраняется не сам агрегат, а только события, связанные с его изменением.

Для хранения данных о событиях в системе используются столбцы Event_id, Event_type и Event_data, которые представляют собой идентификатор события, его тип и описание. В то же время столбцы Aggregate_id и Aggregate_type указывают на идентификаторы и типы соответствующих агрегатов.

Применение архитектурного шаблона Event Sourcing предоставляет ряд преимуществ, основное из которых заключается в автоматической публикации событий при изменении агрегатов.

Это значительно упрощает реализацию событийно-управляемой микросервисной архитектуры.

Данный паттерн позволяет фиксировать полную историю изменений для каждого агрегата, что полезно в случае выполнения временных запросов, позволяющих восстановить предыдущее состояние агрегата.

Тем не менее, даже с учетом всех положительных сторон Event Sourcing, проблема обработки запросов к базе данных остается актуальной. Решением может стать использование паттерна CQRS.

CQRS представляет собой архитектурный подход, который разделяет процессы чтения и обновления данных в хранилище.

Его реализация предполагает деление веб-приложения на командную и запросную части. Командная часть обрабатывает команды, связанные с созданием, чтением, обновлением и удалением агрегатов, а запросная часть занимается обработкой запросов, таких как POST или GET.

В качестве примера можно рассмотреть применение паттерна CQRS в интернет-магазине (рис. 4). Сервисы, отвечающие за управление клиентами и заказами («Customer» и «Order»), выносятся в командную часть приложения, где они предоставляют API-интерфейсы для создания и обновления информации о клиентах и их заказах. В то же время сервис «Viewing customers» находится в запросной части и предоставляет API-интерфейс для получения данных о клиентах.

Этот сервис отслеживает события, исполняемые командной частью, и обновляет данные в хранилище интернет-магазина.

Таким образом, разделение веб-приложения на командную и запросную составляющие повышает производительность, гибкость и безопасность всей системы. Кроме того, с помощью применения паттерна CQRS совместно с событийно-ориентированным подходом решается проблема согласованности данных в микросервисах, позволяя веб-приложению эффективно осуществлять различные виды запросов [12].

Заключение

В статье рассмотрена оптимизация высоконагруженных веб-проектов посредством применения микросервисной архитектуры.

Проведённый анализ показал, что микросервисный подход существенно улучшает гибкость и адаптивность систем за счёт разбиения на автономные модули, что позволяет разработчикам оперативно реагировать на изменяющиеся бизнес-требования и интегрировать новые функциональные элементы без влияния на общую работоспособность системы.

Применение современных технологий, таких как Docker и Kubernetes, способствует автоматизации и оркестрации процессов, что улучшает масштабируемость и снижает время отклика сервисов. Внедрение данных инструментов и подходов позволяет минимизировать риски и обеспечивать отказоустойчивость веб-приложений, что особенно важно для высоконагруженных систем.

Таким образом, микросервисная архитектура представляет собой эффективное решение для создания и оптимизации современных веб-проектов, обеспечивая стабильность и производительность при высоких нагрузках и изменяющихся условиях эксплуатации.

Список литературы:

1. Кравченко Д.А. Микросервисная архитектура //Интерактивная наука. – 2022. – №. 4 (69). – С. 43.
2. Микросервисная архитектура. [Электронный ресурс] Режим доступа: https://vladislaveremeev.gitbook.io/qa_bible/seti-i-okolo-nikh/mikroservisnaya-arkhitektura-microservice-architecture (дата обращения 07.09.2024).
3. Городничев М.Г., Полонский Р.В. Оценка возможности использования микросервисной архитектуры при разработке пользовательских интерфейсов клиент-серверного программного обеспечения // Экономика и качество систем связи. – 2020. – №. 3 (17). – С. 33-43.
4. de Toledo S.S., Martini A., Sjøberg D.I.K. Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study //Journal of Systems and Software. – 2021. – Т. 177. – С. 110968.
5. Söylemez M., Tekinerdogan B., Kolukısa Tarhan A. Challenges and solution directions of microservice architectures: A systematic literature review //Applied sciences. – 2022. – Т. 12. – №. 11. – С. 5507.
6. Кадыров К.А., Сафин А.М. Микросервисная архитектура // Фундаментальные и прикладные научные исследования: актуальные вопросы, достижения и инновации. – 2022. – С. 97-99.
7. Golis T., Dakić P., Vranić V. Automatic Deployment to Kubernetes Cluster by Applying a New Learning Tool and Learning Processes //Proceedings http://ceur-ws. org ISSN. – 2023. – Т. 1613. – С. 0073.
8. Бороздин Н.М. Исследование и анализ практических преимуществ микросервисной архитектуры для современных веб-приложений //Инновационные научные исследования в современном мире. – 2023. – С. 279-284.
9. Abbaspour A. et al. A survey on active fault-tolerant control systems //Electronics. – 2020. – Т. 9. – №. 9. – С. 1513.
10. MSA и не только: как мы создаем высоконагруженные сервисы для банка. [Электронный ресурс] Режим доступа: https://habr.com/ru/companies/vtb/articles/343506/ (дата обращения 07.09.2024).
11. Шнейдеров Е.Н., Фещенко А.А., Боровиков С.М. Организация информационно-компьютерных систем и сетей. Курсовое проектирование: пособие. – 2022.
12. Ганьжа А.Ю., Карелова Р.А. Особенности микросервисной архитектуры событийно-ориентированных веб-приложений // Научное обозрение. Технические науки. – 2021. – № 6. – С. 28-34.