Актуальность применения микросервисной архитектуры в системах обработки данных

Селезнёв Александр Игоревич, студент магистратуры;
Селезнёв Игорь Львович, кандидат технических наук, доцент
Белорусский государственный университет информатики и радиоэлектроники (г. Минск, Беларусь)

В статье рассматриваются основные виды архитектур программного обеспечения. Подробно рассматривается микросервисная архитектура, обсуждаются её достоинства и недостатки. Проведен анализ актуальности применения микросервисной архитектуры в системах обработки данных.

Ключевые слова: *микросервисы, архитектура программного обеспечения, базы данных, программное обеспечение, системы обработки данных.*

Сначала XXI века происходит постоянная модернизация компьютерной инфраструктуры и стремительно меняются требования и потребности IT-компаний, поэтому сложно переоценить важность построения стабильных и производительных систем для обработки различных потоков информации. Успешность и функциональность этих систем во многом обуславливается выбором архитектуры программного обеспечения, которая отвечает поставленным требованиям и задачам отдельно взятой компании. Архитектура программного обеспечения (ПО) — это фундаментальная структура программной системы, которая включает в себя элементы программного обеспечения, отношения между ними, а также свойства как элементов, так и отношений [1].

Основные виды архитектуры ПО в контексте обработки данных

Монолит. Данная архитектура представляет собой систему, в которой весь код развертывается как единый процесс. Может существовать несколько экземпляров этого процесса (из соображений надежности или масштабирования), но весь код упакован в один процесс. На рисунке 1 изображена структурная схема монолита с одним процессом.

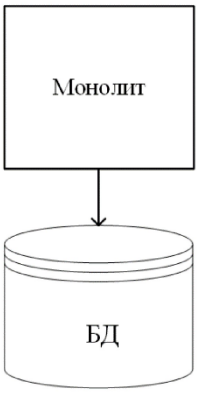

Рис. 1. **Структурная схема монолита с одним процессом**

Из рисунка 1 видно, что рабочее приложение в черном ящике (монолит), выполняющее некоторые операции, имеет доступ к чтению/записи базы данных (БД). Чёрный ящик — это представление системы, внутреннее устройство и механизм работы которой неизвестен или неважен в рамках данной задачи [2]. БД является одним из компонентов системы управления базами данных (СУБД). СУБД представляет собой связанный набор программных компонентов, который позволяет совершать над БД операции чтения/записи, удаления и обработки запросов пользователя. Главным компонентом СУБД является ядро, которое отвечает за работу всей системы [3].

Достоинства монолитной архитектуры:

1. Простота и легкость разработки. Компоненты монолитной системы тесно связаны, поэтому написание и тестирование кода не представляет особой сложности, используется единая база кода, что упрощает понимание общей логики приложения.

2. Производительность и эффективность. Поскольку все компоненты выполняются в рамках одного процесса, взаимодействие между процессами не осуществляется, что обеспечивает более быстрое время выполнения.

3. Среда совместно используемых данных. У всех компонентов есть прямой доступ к БД, что позволяет беспрепятственно обмениваться данными, тем самым устраняя необходимость в сложных механизмах синхронизации.

Недостатки монолита:

1. Масштабирование. Масштабирование монолита представляет собой сложную задачу, потому что всё при-

ложение выступает как единое целое. В случае если выделение дополнительных ресурсов требуется только для определенных компонентов монолита, может произойти их нерациональное использование.

2. Необходимость в отлаженном коде всех внутренних компонентов. Так как монолит потребляет все ресурсы сервера, на котором работает, то в случае использования одним из компонентов значительной части ресурсов может произойти снижение работоспособности всей системы вплоть до отказа.

3. Ограниченность в использовании одного стека технологий. Для добавления новых технологий или фреймворков необходимо перепроектирование всего приложения. Такое требование единообразия технологий может ограничить возможности и замедлить дальнейшее развитие приложения.

4. Необходимость в остановке работы монолита при обслуживании, развертывании или внесении нового функционала. Любые изменения или обновления требуют повторного развертывания всего приложения, что ведет к увеличению времени простоя и возможным нарушением работы [4].

Модульный монолит. Эта архитектура ПО представляет собой систему, в которой один процесс состоит из отдельных модулей. Каждый модуль может работать независимо, но для развертывания модульного монолита эти модули необходимо объединить. На рисунке 2 приведен пример двух основных концепций модульного монолита.

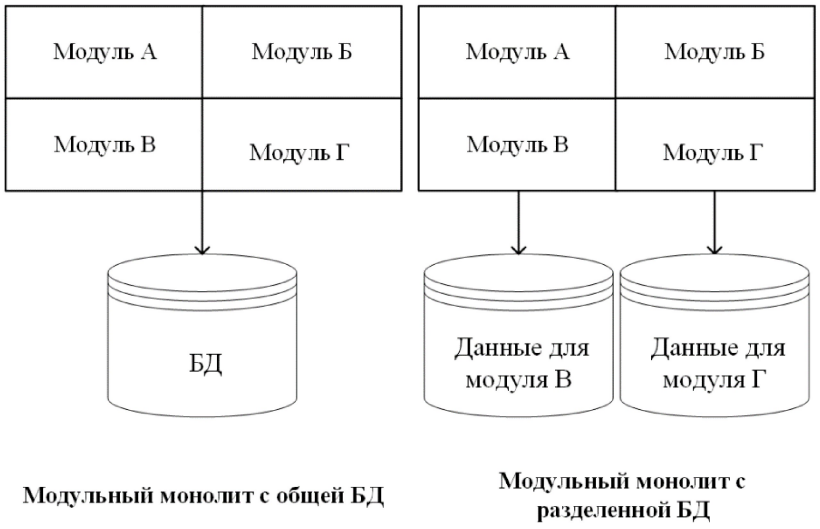

Рис. 2. **Пример основных концепций модульного монолита**

Из рисунка 2 видно, что в случае схемы с общей БД монолит представлен в виде изолированных модулей, которые могут выполнять различные операции и иметь доступ на чтение/запись к БД. Модульный монолит с разделенной БД представляет собой архитектуру, где данные доступны только для определенных модулей. В этом случае, например, для получения данных из модуля Г в модуль Б должен быть прописан интерфейс их взаимодействия, так как данные модуля Г скрыты для всех остальных модулей. В настоящее время модульные монолиты с распределенной БД являются наиболее актуальной версией этой архитектуры.

Модульные монолиты, помимо достоинств обычных монолитов, обладают следующими преимуществами:

1. Изоляция модулей. Модули изолированы друг от друга, поэтому выявление неправильной работы осуществляется легче, чем в обычном монолите. Кроме того, модули, как правило, выполняют одну общую задачу, что облегчает написание и изменение кода.

2. Возможность использования разных стеков технологий для каждого модуля. Так как модули работают изолированно в одной системе, то появляется возможность использовать различные технологии внутри одного модуля при правильно настроенном интерфейсе взаимодействия.

3. Возможность использования разделенной БД для каждого из модулей. Это обеспечивает повышение безопасности использования данных и снижает нагрузку на СУБД.

Несмотря на то что модульные монолиты не имеют ограничений на использование одного стека технологий, тем не менее им присущи все остальные недостатки монолитов, а также добавляется сложность реализации разделения монолита на модули и работа с разделенной БД [5].

Распределенный монолит. Эта архитектура представляет собой систему, состоящую из нескольких сервисов, которые должны быть развернуты единовременно. Распределенный монолит подходит под определение сервис-ориентированной архитектуры (service-oriented architecture, SOA), но не всегда может соответствовать этой архитектуре. SOA представляет собой набор сервисов — независимых, самостоятельных, полноценных приложений, которые могут находиться как на одном хосте, так и в сети [6]. Распределенный монолит, как правило, представляет собой взаимосвязь основного сервиса

и вынесенных во вне отдельных сервисов-приложений, выполняющих множественные задачи, и вся эта система связана между собой. На рисунке 3 представлен пример распределенного монолита.

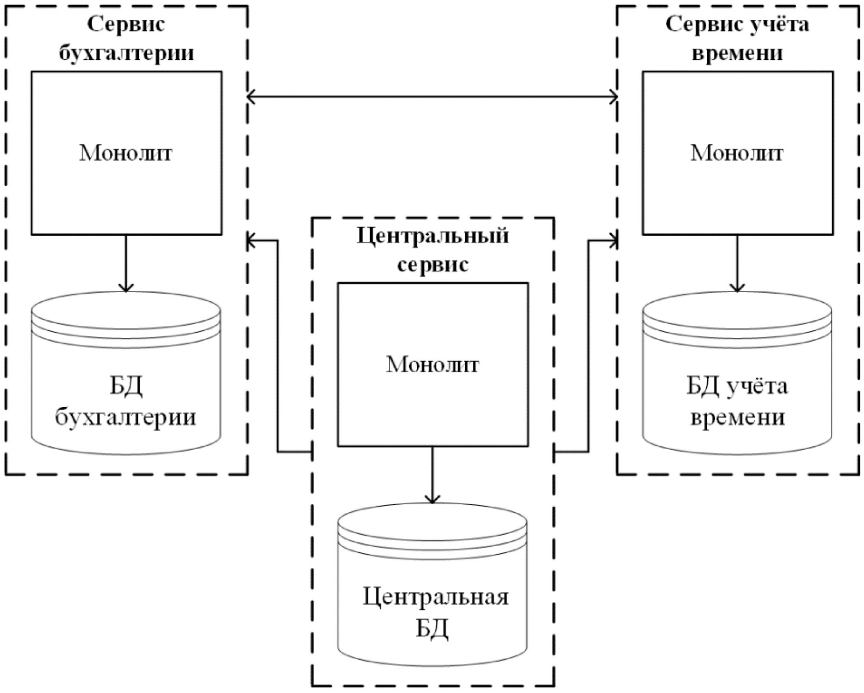

Рис. 3. **Пример распределенного монолита**

Из рисунка 3 видно, что центральный сервис и два других отдельных сервиса взаимодействуют друг с другом посредством интерфейсов, а также имеют отдельные БД.

Несмотря на то, что эта архитектура обладает всеми достоинствами модульного монолита, отдельно вынесенные сервисы не привносят особых достоинств, так как всю систему нужно разворачивать целиком. Главным недостатком реализации распределенного монолита является чрезмерная сложность реализации при увеличении количества сервисов и отсутствие значимых достоинств по сравнению с другими архитектурами.

Микросервисы. Микросервисы — это тип SOA, при котором система строится как набор независимых и слабосвязанных сервисов, которые можно создавать, используя различные языки программирования и технологии хранения данных. Концепция микросервисов позволяет поддерживать слабую связанность сервисов в процессе работы системы и реализует ключевую особенность этой архитектуры — возможность независимого развертывания каждого из микросервисов.

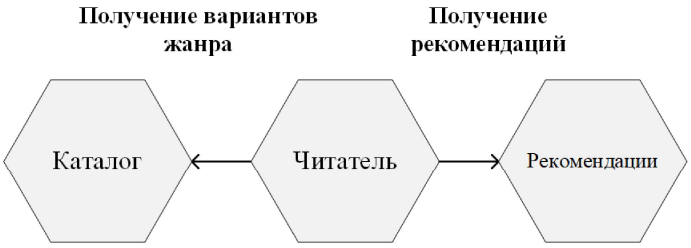

Рис. 4. **Пример микросервисной архитектуры приложения «Чтение онлайн»**

На рисунке 4 приведен пример микросервисной архитектуры (МА) приложения «Чтение онлайн» с тремя микросервисами. Микросервис «Читатель» содержит информацию о пользователе, микросервис «Каталог» позволяет получать доступ к книгам выбранного жанра, представленных в электронном формате, и, наконец, микросервис «Рекомендации» на основе полученных запросов пользователя формирует рекомендации в рамках текущего жанра.

Микросервисы более детально рассматриваются ниже.

Особенности микросервисной архитектуры

Ключевая особенность микросервисов — возможность реализации системы, при которой каждый из компонентов может быть развернут независимо от других, что достигается с помощью систем непрерывной интеграции, непрерывной доставки и непрерывного развертывания.

Непрерывная интеграция (Continuous Integration, CI) — это система, состоящая из системы контроля версий (например Git) и сервера, который включает в себя модули сборки и модульного тестирования. Особенностью CI является автоматизация всего процесса, на выходе которого получается работающий код.

Непрерывная доставка (Continuous Delivery, CD) представляет собой систему, которая реализует подготовку тестовой среды и интеграционных тестов. Различают следующие тестовые среды:

1. Среда для разработки. В этой среде производятся тесты с меньшим количеством компонентов и библиотек; используются только те, которые необходимы для работы приложения.
2. Среда для ручного тестирования. В этой среде тестируется любое возможное взаимодействие между приложением и пользователем.
3. Среда для нагрузочного тестирования. Эта среда предназначена для тестирования больших объемов данных и интенсивного использования приложения.
4. Среда, повторяющая реальные параметры работы приложения в ожидаемых и прогнозируемых условиях. В ней, как правило, идет тестирование отдельных параметров.
5. Среда, максимально приближенная по всем параметрам к реальной среде, в которой работает или будет работать приложение.

После полного выполнения всех тестов код в любой момент готов к развертыванию, что является базисом CD.

Непрерывное развертывание (Continuous Deployment, CDep) — это процесс выпуска ПО, в котором используется автоматизированное тестирование и производится проверка правильности и стабильности изменений в базе кода для немедленного автономного развертывания в рабочей среде. В отличии от CD и CI, которые, как правило, автоматизируются, в CDep предусматривается как автоматический, так и ручной процесс отправления ПО для использования [7].

Схема взаимодействия систем CI, CD и CDep представлена на рисунке 5.

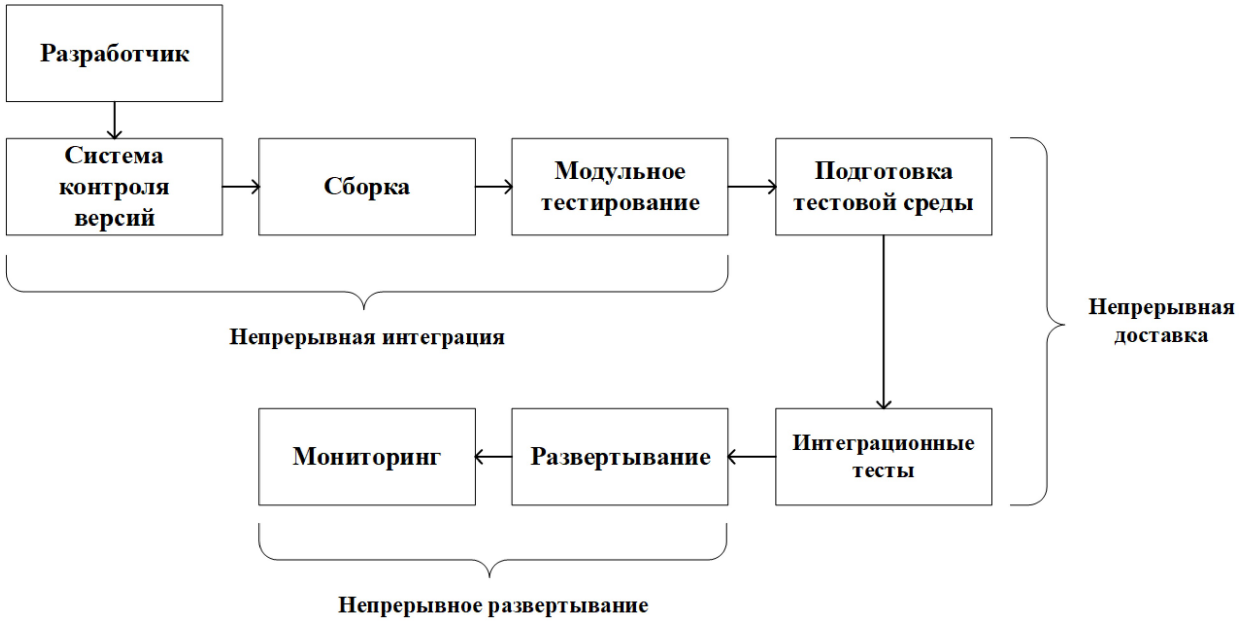

Рис. 5. **Схема взаимодействия систем CI, CD и CDep**

Для реализации данного подхода микросервисы должны обладать как можно большей связностью и как можно меньшей сопряженностью. Связность — организация функциональности кода микросервиса таким образом, чтобы обеспечить возможность вносить изменения в меньшем количестве мест. Это нужно для того, чтобы система микросервиса оставалась целостной при любых изменениях. Сопряженность — это уровень взаимодействия между модулями. Когда между модулями присутствует слабая сопряженность, то изменения в одном микросервисе не требуют изменений в другом. Слабо сопряженный микросервис имеет необходимый минимум сведений о сервисах, с которыми ему приходится работать. На рисунке 6 представлены различные виды сопряженности микросервисов.

Доменная (предметная) сопряженность описывает взаимодействие микросервисов, когда одному микросервису требуется функциональность, предоставляемая другим. Доменная форма связи считается наиболее слабой. Ситуация, когда микросервису необходимо взаимодействовать с большим количеством нижестоящих микросервисов, может свидетельствовать о чрезмерной централизованности логики. Временная сопряженность — это зависимость работы одного микросервиса от другого в определённый временный интервал. Сквозная сопряженность характеризуется тем, что для работы одного микросервиса требуются данные от другого микросервиса, который не имеет с ним прямого интерфейса взаимодействия и получает их через посредников — другие

Рис. 6. **Виды сопряженности микросервисов**

микросервисы. Общая сопряженность возникает тогда, когда несколько микросервисов используют общий набор данных. Одним из видов такой формы сопряженности является множество микросервисов, использующих одну и ту же БД, это также может проявляться в использовании общей памяти или файловой системы. Сопряженность по содержимому проявляется, когда вышестоящий сервис взаимодействует с внутренними компонентами нижестоящего и изменяет его состояние. Примером этого является ситуация, когда один микросервис обращается к БД другого микросервиса и производит её изменения напрямую. Сопряженность по содержимому во многом похожа на общую сопряженность, но отличается от неё большим размытием границ доступа между микросервисами, что может повлечь за собой пропорциональное увеличение сложности изменений системы в целом. Проблема в том, что изменения в одном микросервисе могут вызвать нарушения в работе других микросервисов, а это противоречит основной концепции данной архитектуры [8].

Для полноценной работы МА важным требованием является сокрытие информации в рамках отдельно взятого микросервиса. Основной способ реализации данного требования заключается в отделении частей кода, подверженных частым изменениям, от тех, которые являются статичными. Это нужно для того, чтобы граница микросервиса (интерфейсы взаимодействия с внешним миром/микросервисами) была стабильной, а внутренние модифицируемые части модуля были скрыты. Идея заключается в том, что внутренние изменения могут вноситься безопасно до тех пор, пока поддерживается совместимость модулей.

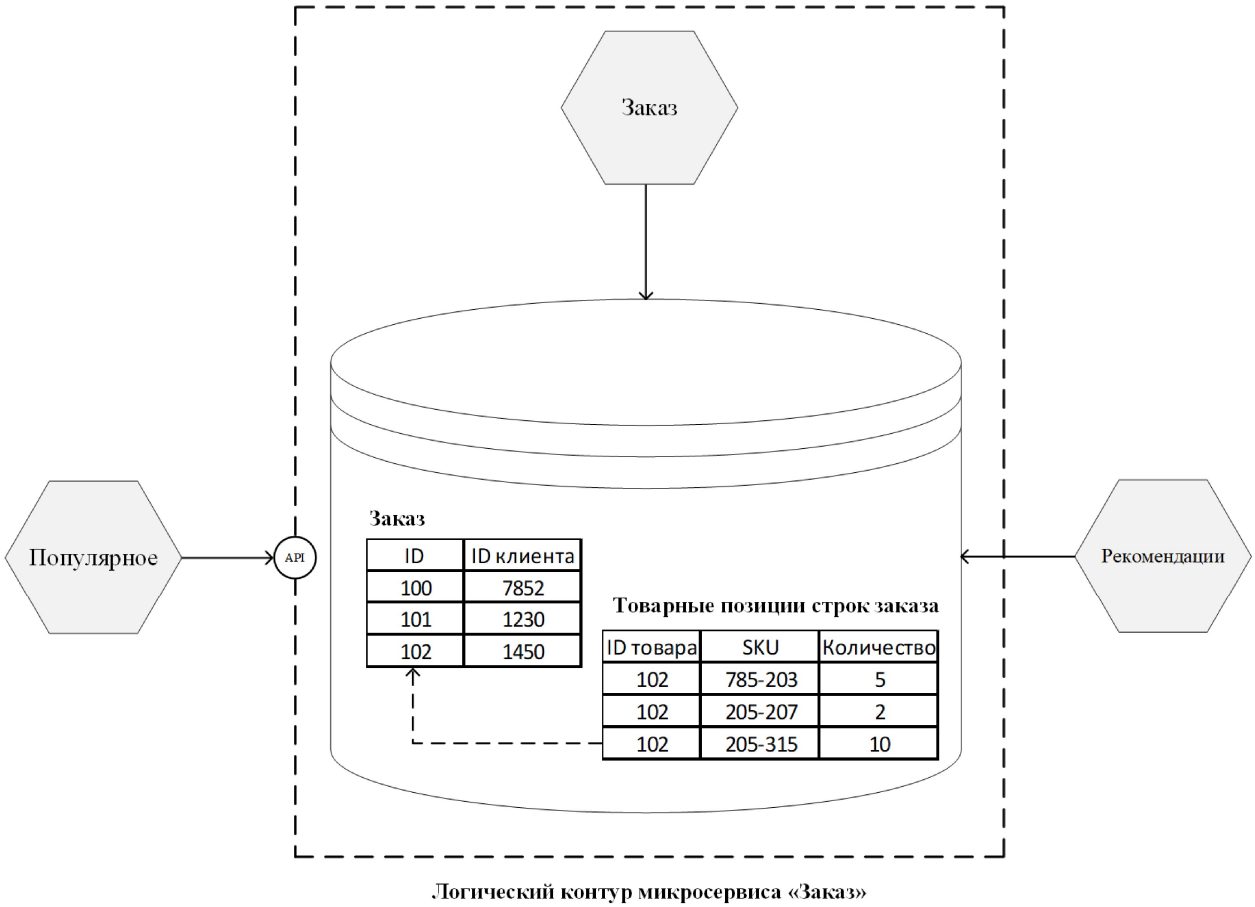

Рис. 7. **Пример трех микросервисов с различным сокрытием информации**

На рисунке 7 представлена система обработки заказа с помощью микросервисов «Заказ», «Рекомендации» и «Популярное». После исполнения операций в модуле «Заказ» данные вносятся в его БД, к которой обращаются микросервисы «Рекомендации» и «Популярное». Микросервис «Рекомендации» обращается к БД микросервиса «Заказ» напрямую и извлекает необходимые данные, таким образом эти два микросервиса имеют общий тип сопряженности и слабое сокрытие информации. В этом случае присутствует сложный канал передачи информации, так как одному микросервису нужно «знать» внутреннее устройство другого микросервиса и способ извлечения данных из него — в случае изменения первоначального микросервиса другой перестанет корректно функционировать. Микросервис «Популярное» обращается к микросервису «Заказ» через программный интерфейс (Application Programming Interface, API), представляющий собой конечную точку микросервиса «Заказ». В данном случае API представляет собой подобие контракта между этими микросервисами: микросервис «Заказ» ожидает предоставления данных на входе в форме запроса на чтение данных из его БД, обязуется выполнить внутреннюю операцию после получения корректного запроса и передать данные на выход. В этом примере выполняется операция чтения данных из БД и отправка их копий в микросервис «Популярное» [9], что обеспечивает сокрытие данных и реализацию доменной сопряженности. Для построения стабильной и гибкой системы на микросервисах справедливо утверждение «умные конечные точки и глупые каналы передачи», подразумевающее организацию функционально сложных входных точек микросервисов, например через API, и простые каналы обмена данными [10]. На примере рисунка 7 можно утверждать о непосредственной взаимосвязи организации БД по отношению к микросервисам.

Реализация МА предполагает следующие виды взаимодействия с СУБД:

Использование одной СУБД и разделение базы данных на логические схемы БД для каждого из микросервисов. Возможность разделения общей БД на отдельные логические схемы, которые существуют обособленно и изолированно по отношению друг к другу, отвечает требованиям микросервисной архитектуры. Ядро СУБД в данном случае представляет собой единую точку отказа системы, что, как правило, повышает отказоустойчивость, уменьшает сложность проектирования и затраты на лицензирование.

Использование нескольких СУБД с предоставлением микросервисам своих отдельных БД и/или логических схем. В зависимости от реализации это может расширить функциональность системы, но значительно повышает сложность поддержки, проектирования и создает дополнительные точки отказа системы. В случае отказа ядра одной из СУБД остальная часть системы может некорректно функционировать, так как ряд микросервисов не сможет получать данные из отказавших БД.

Для достижения гибкости в распределении рабочей нагрузки на физическом уровне в МА присутствует возможность динамического изменения используемых ресурсов каждым микросервисом с помощью инстанцирования дополнительных экземпляров этого микросервиса и балансировщика нагрузки в качество элемента управления (рис. 8).

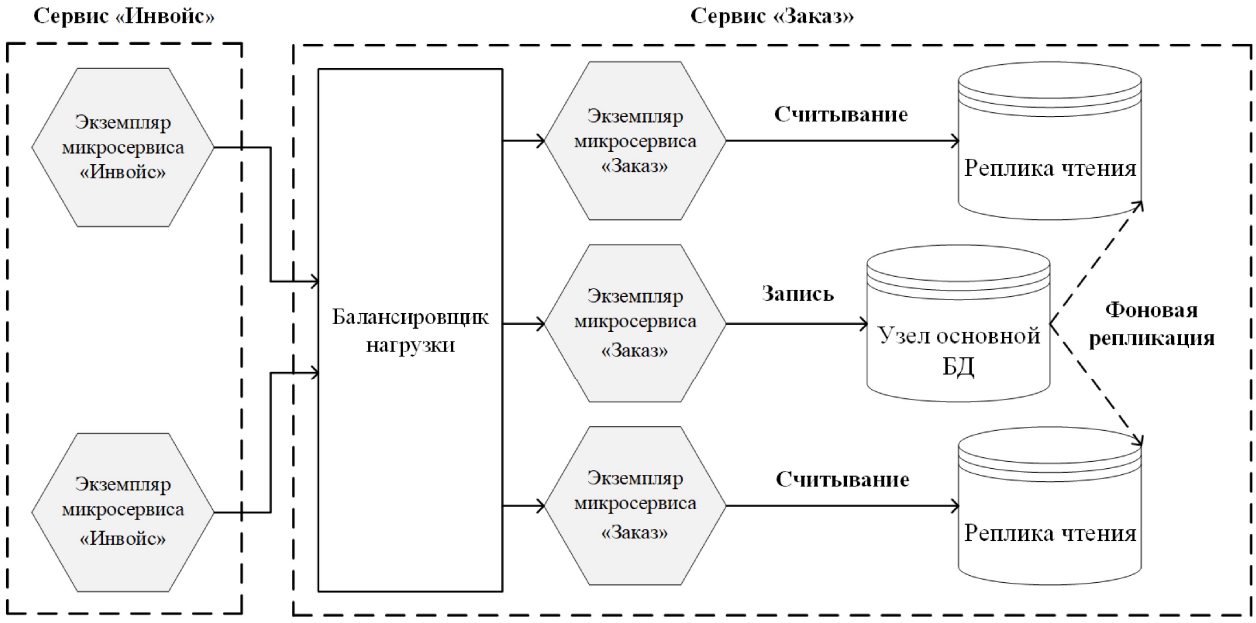

Рис. 8. **Пример взаимодействия микросервисов на физическом уровне**

Из рисунка 8 видно, что физическое представление взаимодействия двух микросервисов существенно отличается от логического. При изменении рабочей нагрузки возможно подключение нескольких экземпляров микросервиса «Инвойс», которые обращаются к определенным экземплярам микросервиса «Заказ» под управлением ба-

лансировщика нагрузки. Дополнительные экземпляры микросервиса позволяют обрабатывать большую нагрузку системы и увеличивают её надежность — в случае отказа одного экземпляра может быть использован другой. В рассматриваемом примере для обработки нагрузки выделены 3 экземпляра микросервиса «Заказ», которые производят операции чтения/записи в узел основной БД. Современные СУБД предоставляют возможность создавать реплики для чтения БД, каждая из которых используется отдельным экземпляром микросервиса, что приводит к уменьшению нагрузки на используемую БД, так как она может быть одной из многих логических схем, выделенных под каждый микросервис.

Основные варианты развертывания микросервисов:

1. Физический компьютер. В этом случае весь микросервис развертывается на локальной машине разработчика, представляющей собой стационарный компьютер. Несмотря на возможность полного доступа к управлению микросервисом, этот метод обладает рядом недостатков: нерациональное использование ресурсов, нарушение принципа «изолированная среда для выполнения» и сложность в управлении. В настоящее время этот метод почти не используется.

2. Виртуальные машины. Виртуальная машина (ВМ) — это программное обеспечение, которое создает виртуальное окружение, эмулирующее работу физического компьютера или сервера [11]. ВМ позволяет изолировать микросервисы и повысить эффективность использования вычислительной инфраструктуры — например, в составе облачных услуг. Развертывание каждого экземпляра микросервиса на отдельной ВМ гарантирует хорошую степень изоляции между экземплярами. Недостатком такой конфигурации является то, что любая система управления ВМ (гипервизор) также потребляет часть аппаратных ресурсов, причем пропорционально количеству ВМ, что сильно ограничивает дальнейшее масштабирование.

3. Контейнеры. Контейнер — это альтернативный тип виртуализации, использующий изолированную среду для выполнения приложений [12]. Ключевым преимуществом микросервисов, упакованных в контейнеры, по сравнению с ВМ, является их сравнительно небольшой размер и высокая скорость запуска, что повышает эффективность динамического перераспределения нагрузки. Недостатком контейнеров для микросервисов является меньшая степень изоляции, чем на физической машине и ВМ.

Контейнеры могут использоваться в совокупности с ВМ, позволяя получать преимущества обоих видов развертывания: ВМ обеспечивают высокую производительность и большую степень изоляции, а контейнеры — высокую скорость развертывания и работы. Популярными контейнерными технологиями являются Docker и Kubernetes.

4. Платформа как услуга (Platform as a Service, PaaS). PaaS — это уровень облачной инфраструктуры, предоставляющий ресурсы для создания инструментов и приложений уровня пользователя. В него входит базовая инфраструктура, включая вычислительные и сетевые ресурсы, хранилища, а также инструменты разработки, СУБД и промежуточное ПО [13]. Особенностью PaaS для развертывания МА является то, что её инструменты, как правило, представляют собой черные ящики и адаптированы к среднестатистическому приложению. В зависимости от потребностей для текущей системы это может быть как достоинством, так и недостатком. Пример такого решения — платформа Heroku [14].

5. Функция как услуга (Function as a Service, FaaS). FaaS — архитектурный шаблон, предполагающий возможность вызова экземпляра управляющего кода без необходимости управления серверами и серверным приложением; является ключевым компонентом вычислений без сервера [15]. Одним из популярных решений FaaS является сервис AWS Lambda [16], концепция которого состоит в том, что система строится из модулей рабочего кода (функций) и триггеров, которые могут быть в виде файлов, событий или вызовов. После того как поступает нужный сигнал — триггер активируется, и функция запускается, а когда сигнал прекращается — она завершается. Преимуществом такой системы является гибкая модель оплаты за использование: в отличие от других систем, где нужно оплачивать целиком использования всего сервиса, в этой системе оплачиваются только те функции, которые требуются.

FaaS является хорошим решением для ситуаций, когда нагрузка на микросервисную систему низкая или имеет нелинейных характер. Основным преимуществом FaaS является снижение операционных издержек, а недостатком — ограниченность в контроле над запускаемыми процессами.

Достоинства микросервисной архитектуры:

1. Технологическая неоднородность. Для реализации каждого микросервиса может быть выбран свой стек технологий, который оптимизирует и улучшает работу системы в целом. Ввиду вышеописанных особенностей МА есть возможность во внедрении и проверке новых технологий без отказов в работе.

2. Надежность. При отказе одного из элементов есть возможность в продолжении работы всей системы с помощью изоляции этого элемента, в отличии от монолита, где, как правило, отказавший сервис приводит к отказу всей системы.

3. Масштабирование. Особенностью МА является гибкость по отношению к рабочим нагрузкам в любом месте системы, так как есть возможность динамического увеличения ресурсов определенного микросервиса с помощью запуска дополнительных экземпляров.

4. Простота развертывания. МА позволяет производить развертывание как отдельного фрагмента кода микросервиса, так и всего микросервиса без надобности пересборки всей системы и остановки работы приложения. Для этого, как правило, применяются системы CI, CD и CDep.

5. Компонуемость. Концептуально микросервисы соответствуют утверждению «один микросервис — одна задача», что дает возможность как повторного использования отдельных микросервисов, так и перекомпоновки всей системы.

Недостатки микросервисной архитектуры:

1. Сложность/невозможность в локальном запуске всей системы. По мере увеличения количества используемых микросервисов ограничивается возможность локального тестирования и разработки системы в целом.

2. Технологическая перегруженность. Использование микросервисов вносит множество затруднений и проблем, поэтому добавление и внедрение дополнительных стеков технологий, особенно одновременно, может значительно ухудшить имеющуюся систему.

3. Стоимость. Микросервисы являются «дорогой» технологией, так как требуется правильное и сбалансированное проектирование системы, опыт разработчиков и внедрение/лицензирование новых технологий.

4. Отчетность. МА представляет собой разнородную систему взаимодействия различных элементов с изолированными БД, поэтому, в отличие от монолита, в котором есть возможность считывания всей БД целиком и простого составления отчетности, необходимо настраивать отдельную систему для сбора информации.

5. Мониторинг и устранение неполадок. Мониторинг этой архитектуры представляет собой сложную задачу, так как на физическом уровне каждый микросервис состоит из некоторого количество экземпляров — активных и неактивных, а также БД, с некоторым количеством реплик, что значительно усложняет мониторинг системы в целом.

6. Тестирование. Сложность тестирования выражается в том, что сначала нужно отдельно тестировать каждый микросервис, а потом совместное взаимодействие его с другими элементами системы.

7. Согласованность данных. Каждый микросервис обладает своей БД, поэтому возникает проблема согласованности данных при передаче, так как заложенные механизмы транзакций СУБД хорошо работают только в рамках одной БД [17, с. 47–57].

Актуальность применения микросервисной архитектуры

В зависимости от процесса разработки/модернизации ПО в IT-компании могут применяться два вида внедрения МА:

1. Выбор микросервисов в качестве исходной архитектуры при создании нового приложения. В ходе разработки нового приложения предметная область может изменяться. Такие изменения при использовании МА могут приводить к большему количеству преобразований, вносимых в интерфейсы сервисов, а их взаимная координация требует выделения дополнительных ресурсов. Поэтому микросервисы целесообразно использовать только в случае явно определенной предметной области.

С другой стороны, при определенной предметной области также могут возникнуть трудности. Основная часть этих затруднений связана с необходимостью параллельной разработки основного функционала приложения и внедрения МА. Еще одной сложностью является то, что микросервисы во многом ориентированы на облачную инфраструктуру, которая может вносить свои коррективы в разработку. В этом случае монолит может быть более предпочтителен, позволяя сосредоточиться именно на разработке приложения.

2. Изменение исходной архитектуры. Под изменением исходной архитектуры, как правило, понимается разложение той или иной реализации монолита. Разложение монолита — это процесс декомпозиции системы на меньшие функциональные части, каждая из которых, в контексте микросервисов, предполагает собой реализацию одной обособленной функцией. В зависимости от потребностей конкретной IT-компании могут быть сделаны промежуточные шаги на пути принятия МА: преобразование в модульный монолит или переход к распределенному монолиту. Примером перехода к модульному монолиту является ПО Shopify, которое в итоге настолько отвечало требованиям компании, что дальнейший переход на микросервисы не потребовался [18].

Начало разложения монолита, как правило, подразумевает собой выделение его функциональных частей, определение их взаимосвязей и поиск тех элементов, декомпозиция которых будет наиболее проста в реализации и увеличит эффективность работы приложения [19, с. 77–83]. Для полной декомпозиции необходимо извлечь как отдельные модули (фрагменты кода), так и разделить общую базу данных на обособленные и изолированные БД.

На рисунке 9 представлен пример начального этапа разложения монолита «Библиотека онлайн», в котором были выделены следующие функциональные модули: «Отчётность», «Популярное», «Реклама», «Читатель» и «Каталог». В этом монолите основным функциональным блоком является «Читатель», в котором реализованы функции работы приложения. В модуле «Отчётность» заносятся отчёты о пользовании приложением, модули «Каталог» и «Популярное» служат для отбора книг из каталога и занесения в популярные книги соответственно, а модуль «Реклама» отвечает за показ рекламного объявления о популярных книгах на домашней странице пользователя. Так как модуль «Отчётность» имеет меньше всего взаимосвязей и регулярно используется для сбора информации, то целесообразно начать разложение монолита с него путем выделения в отдельный микросервис. В результате этой операции для выделенного микросервиса можно ограниченно начать пользоваться некоторыми преимуществами микросервисов: динамически изменять выделение ресурсов и использовать другой стек технологий для более эффективной обработки данных. В некоторых случаях такой частичный переход к МА может помочь разгрузить запутанную вну-

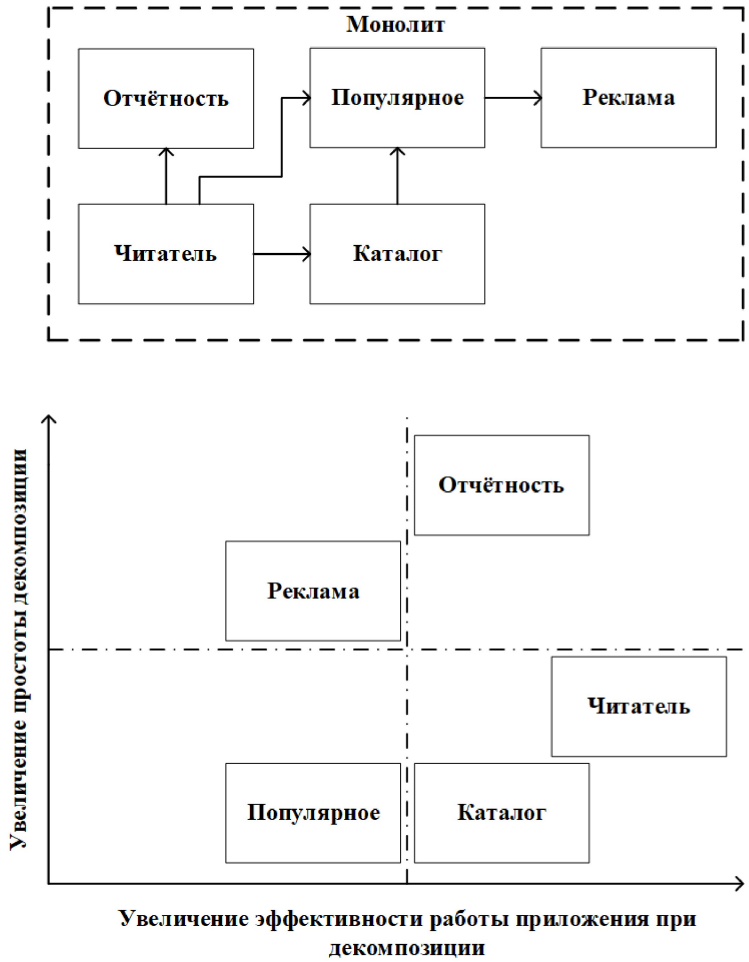

Рис. 9. **Пример начального этапа разложения монолита «Библиотека онлайн»**

треннюю логику отдельных сервисов приложения и явно изолировать важные области данных, доступ к которым должен быть ограничен. Однако, чем больше взаимосвязей имеет функциональный модуль монолита, тем сложнее процесс его выделения в микросервис: к таким элементам относятся функциональные модули «Читатель» и «Популярное». С другой стороны, как правило, выделение таких модулей в отдельные микросервисы может значительно повысить производительность работы всей системы.

Для каждого выделенного функционального модуля необходимо создать отдельную логическую схему БД путем разделения исходной базы данных монолита, что является сложной задачей, поскольку для нескольких БД невозможно гарантировать ACID транзакции, в отличии от работы с одной БД, где они реализуются программно на уровне СУБД. ACID (atomicity, consistency, isolation, durability) — это набор требований к транзакционной системе, обеспечивающий наиболее надёжную и предсказуемую работу: атомарность, согласованность, изоляция и устойчивость. Такое представление общей структуры данных в виде логических схем БД позволяет значительно увеличить безопасность и конфиденциальность, так как будет устранена возможность их неявного изменения.

Переход от монолитного приложения к полноценной МА открывает новые возможности для разработчиков, если они смогут совладать со всеми трудностями этой архитектуры. Многие крупные IT-компании успешно мигрировали от монолита в микросервисную архитектуру: PayPal, eBay, Amazon, Airbnb и другие.

Выводы

Микросервисы популярны ввиду своих достоинств, ключевыми из которых являются высокая масштабируемость и возможность обособленного изменения отдельных модулей приложения без простоев в работе. Однако, для получения максимальной отдачи от этой архитектуры требуются значительные финансовые вложения, большой опыт работы с различными стеками технологий и однозначное представление того, зачем переходить именно на микросервисы. В ином случае оправданным и действенным методом может быть переход к модульному монолиту с возможностью дальнейшего мигрирования на микросервисы.

Несмотря на то, что микросервисы являются инновационной и развитой технологией, к ним следует подходить взвешенно при выборе в качестве главной архитектуры.

Литература:

1. Разработка архитектуры для чайников. Часть 1. — Текст: электронный // Habr: [сайт]. — URL: https://habr.com/ru/articles/658145/ (дата обращения: 28.11.2023)
2. What Is a Black Box Model? Definition, Uses, and Examples. — Текст: электронный // Investopedia: [сайт]. — URL: https://www.investopedia.com/terms/b/blackbox.asp (дата обращения: 29.11.2023).
3. Система управления базами данных: что это такое и зачем она нужна. — Текст: электронный // Skillbox: [сайт]. — URL: https://skillbox.ru/media/code/sistema-upravleniya-bazami-dannykh-chto-eto-takoe-i-zachem-ona-nuzhna/ (дата обращения: 29.11.2023).
4. Monolith vs Microservice Architecture: A Comparison. — Текст: электронный // Camunda: [сайт]. — URL: https://camunda.com/blog/2023/08/monolith-vs-microservice-architecture-comparison/ (дата обращения: 28.11.2023).
5. Модульный монолит вместо микросервисов: как, когда и зачем. — Текст: электронный // Ctfassets: [сайт]. — URL: https://assets.ctfassets.net/9n3x4rtjlya6/6Gx9h6AsXvJFHtYyPuu5wI/9035856ac0b7f7786a8a75955ab94cee/_____-_____.pdf (дата обращения: 28.11.2023).
6. Сервисно-ориентированная архитектура (SOA). — Текст: электронный // Блог программиста: [сайт]. — URL: https://pro-prof.com/forums/topic/service-oriented-architecture (дата обращения: 28.11.2023).
7. Зачем нужны непрерывная доставка и непрерывное развертывание? — Текст: электронный // Habr: [сайт]. — URL: https://habr.com/ru/companies/custis/articles/557540/ (дата обращения: 28.11.2023).
8. Моделирование микросервисов. Часть 1. — Текст: электронный // Habr: [сайт]. — URL: https://habr.com/ru/articles/742206/ (дата обращения: 28.11.2023).
9. Что такое API. — Текст: электронный // Habr: [сайт]. — URL: https://habr.com/ru/articles/464261/ (дата обращения: 28.11.2023).
10. 25 вопросов о микросервисах, на которые вы, скорее всего, не сможете ответить. — Текст: электронный // Senior: [сайт]. — URL: https://senior.ua/articles/25-voprosov-o-mikroservisah-na-kotorye-vy-skoree-vsego-ne-smozhete-otvetit (дата обращения: 29.11.2023).
11. Что такое виртуальная машина и гипервизор и зачем они нужны. — Текст: электронный // Beseller: [сайт]. — URL: https://beseller.by/blog/virtualnyye-mashiny-gipervizory/ (дата обращения: 28.11.2023).
12. Селезнёв, А. И. Контейнеризация в системах обработки данных / А. И. Селезнёв, И. Л. Селезнёв. — Текст: непосредственный // Молодой учёный. — 2023. — № 43. — с. 7–11.
13. Платформа как сервис. — Текст: электронный // Atlassian: [сайт]. — URL: https://www.atlassian.com/ru/microservices/cloud-computing/platform-as-a-service (дата обращения: 28.11.2023).
14. Что такое Heroku? Все секреты раскрыты…. — Текст: электронный // Back4App: [сайт]. — URL: https://blog.back4app.com/ru/что-такое-heroku/ (дата обращения: 28.11.2023).
15. Function as a service (FaaS). — Текст: электронный // Techtarget: [сайт]. — URL: https://www.techtarget.com/searchitoperations/definition/function-as-a-service-FaaS (дата обращения: 29.11.2023).
16. AWS Lambda — теория, знакомство. — Текст: электронный // Habr: [сайт]. — URL: https://habr.com/ru/articles/457100/ (дата обращения: 29.11.2023).
17. Ньюмен, Сэм. Создание микросервисов / Сэм Ньюмен. — 2-е изд. — Санкт-Петербург: ООО «Прогресс книга», 2023. — 624 с. — Текст: непосредственный.
18. Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity. — Текст: электронный // Shopify: [сайт]. — URL: https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity (дата обращения: 28.11.2023).
19. Ньюмен, Сэм. От монолита к микросервисам / Сэм Ньюмен. — 1-е изд. — Санкт-Петербург: ООО «БХВ-Петербург», 2021. — 272 с. — Текст: непосредственный.