**Архитектурные паттерны микросервисов**

**Монолитная архитектура** — это традиционная модель разработки программного обеспечения, в которой одна база кода используется для выполнения нескольких бизнес-функций. Все программные компоненты монолитной системы взаимозависимы из-за использования встроенных механизмов обмена данными внутри системы. Модификация монолитной архитектуры возможна лишь частично и занимает много времени, поскольку даже небольшие изменения затрагивают большие области базы кода.

Напротив, архитектурный подход на основе **микросервисов** подразумевает, что программное обеспечение состоит из небольших независимых компонентов (сервисов). Каждый сервис выполняет одну функцию и взаимодействует с другими сервисами через четко определенные интерфейсы. Поскольку они работают независимо, вы можете обновлять, изменять, развертывать или масштабировать каждую службу отдельно по мере необходимости.

|  |  |  |
| --- | --- | --- |
| **Категория** | **Монолитная архитектура** | **Архитектура микросервисов** |
| Проектирование | Единая кодовая база с несколькими взаимозависимыми функциями. | Независимые программные компоненты с автономными функциональными возможностями, которые взаимодействуют друг с другом через API. |
| Разработка | Требует меньше планирования на начальном этапе, но сложность понимания и поддержки постепенно растет. | Требует больше планирования и инфраструктуры на начальном этапе, но со временем управление и обслуживание упрощаются. |
| Развертывание | Все приложение развернуто как единое целое. | Каждый микросервис представляет собой независимый программный объект, требующий индивидуального контейнерного развертывания. |
| Отладка | Весь путь кода можно отслеживать в одной среде. | Требуются сложные инструменты отладки, умеющие отслеживать обмен данными между несколькими микросервисами. |
| Модификация | Небольшие изменения влекут за собой большие риски, поскольку затрагивают всю базу кода. | Можно изменять отдельные микросервисы, не затрагивая приложение в целом. |
| Масштабирование | Необходимо масштабировать все приложение, даже если увеличится нагрузка только на некоторые функциональные области. | При необходимости можно масштабировать отдельные микросервисы, что снижает общие затраты на масштабирование. |
| Инвестиции | Низкие первоначальные инвестиции, но более высокий объем работ по текущему и техническому обслуживанию. | Дополнительные затраты времени и средств на создание инфраструктуры и накопление опыта в команде. С другой стороны — долгосрочная экономия затрат, более простое обслуживание и адаптивность. |

**Паттерн Strangler**

Этот шаблон означает миграцию монолитного приложения на микросервисную архитектуру путем постепенного переноса существующих функций в микросервисы. Настраивается маршрутизация запросов между устаревшим монолитом и микросервисами. Когда очередная функциональность переносится из монолита в микросервисы, фасад перехватывает клиентский запрос и направляет его к микросервисам. Новые функции при этом реализуются исключительно в микросервисах, минуя монолит. После переноса всех функций монолитное приложение полностью выводится из эксплуатации.

Паттерн микросервиса Strangler имеет свои плюсы и минусы:

Плюсы

* Легче включить новые требования в качестве нового сервиса.
* Двухэтапный подход облегчает создание микросервиса, монолита в модульный монолит.
* Безопасный подход, поскольку старые сервисы продолжают работать с новыми отрефакторенными сервисами.

Минусы

* Разделять приложение на модули непросто, если компоненты тесно связаны друг с другом, требуется много времени на разработку, а также увеличиваются усилия по тестированию.
* План отката должен быть под рукой на случай, если что-то пойдет не так во время рефакторинга. По опыту автора, производственная среда/сценарии совершенно разные.
* Увеличение затрат на инфраструктуру.

**API-шлюз**

— это сервис, который служит точкой входа в приложение из внешнего мира. Он отвечает за маршрутизацию запросов, объединение API и другие возможности, например аутентификацию. В данном разделе мы рассмотрим его со всеми преимуществами и недостатками. Вы познакомитесь с проблемами проектирования, которые необходимо решить при разработке API-шлюза. API-шлюз отвечает за маршрутизацию запросов, объединение API и преобразование протоколов. Все запросы, выполняемые внешними клиентами, сначала поступают на API-шлюз, который направляет их подходящим сервисам. Для обработки других запросов API-шлюз использует объединение API, обращаясь к разным сервисам и агрегируя полученные результаты. Он может также налаживать связь между клиентскими протоколами, такими как HTTP и WebSockets, и внутренними протоколами сервисов, плохо совместимыми с клиентами. API-модуль реализует каждую API-операцию одним из двух способов. Некоторые операции накладываются непосредственно на определенную API-операцию сервиса, которой впоследствии направляются соответствующие запросы. Перенаправляются запросы с помощью универсального модуля, который считывает конфигурационный файл с правилами маршрутизации. Чтобы реализовать более сложные API-операции, API-модуль использует объединение API. Это требует написания дополнительного кода. Каждая реализация API-операции обрабатывает запросы, обращаясь к нескольким сервисам и объединяя результаты.

**Преимущества API-шлюза**

Большое преимущество от использования API-шлюза связано с тем, что он инкапсулирует внутреннюю структуру приложения. Вместо вызова определенных сервисов клиенты общаются со шлюзом. Каждый клиент получает отдельный API, что снижает количество запросов между ним и приложением. К тому же это упрощает клиентский код.

**Недостатки API-шлюза**

Шаблон API-шлюза имеет определенные недостатки. Это еще один высокодоступный компонент, который нужно разрабатывать, развертывать и администрировать. Вдобавок существует риск того, что API-шлюз начнет тормозить разработку. Его следует обновлять при «выставлении наружу» API очередного сервиса. Важно, чтобы процесс обновления был максимально легковесным. В противном случае разработчики будут вынуждены ждать своей очереди, чтобы обновить API-шлюз. Несмотря на это, шаблон подходит для большинства реальных приложений. При необходимости можно применить его, чтобы команды разрабатывали и развертывали свои API независимо друг от друга.

**Три проблемы, которые часто встречаются при реализации запросов в микросервисной архитектуре.**

* Объединение API для извлечения данных, разбросанных по разным сервисам, приводит к затратным малоэффективным операциям JOIN, выполняемым в памяти.
* Сервис, владеющий данными, хранит их в формате или базе данных, которые не имеют эффективной поддержки нужного запроса.
* Необходимость разделения ответственности означает, что реализацией запроса должен заниматься не тот сервис, который владеет данными. Шаблон CQRS решает все эти проблемы.

CQRS расшифровывается как разделение ответственности командных запросов. Как следует из названия, этот шаблон предназначен для разделения обязанностей. У обеих версий сервиса (с CQRS и без него) есть API, состоящий из различных CRUD-операций. В сервисе, не основанном на CQRS, эти операции обычно реализуются доменной моделью, привязанной к базе данных. Для улучшения производительности некоторые запросы могут миновать доменную модель и обращаться к базе данных напрямую. Единая хранимая модель данных поддерживает и команды, и запросы. В сервисе, основанном на CQRS, доменная модель командной стороны обрабатывает СRUD-операции и привязана к собственной базе данных. Она может обрабатывать также простые запросы, использующие первичные ключи и не содержащие операций слияния. Командная сторона публикует события при каждом изменении своих данных. Для этого может задействоваться фреймворк, такой как Eventuate Tram, или порождение событий. За нетривиальные запросы отвечает отдельная модель. Она намного проще по сравнению с командной стороной, потому что ей не нужно реализовывать бизнесправила. Чтобы поддерживать необходимые запросы, эта модель использует подходящую для этого базу данных. Она содержит обработчики, которые подписываются на доменные события и обновляют базу (-ы) данных. Таких моделей может быть несколько, по одной для каждого вида запросов.

**Преимущества CQRS**

* Возможность эффективной реализации запросов в микросервисной архитектуре.
* Возможность эффективной реализации разнородных запросов.
* Возможность выполнения запросов в приложении, основанном на порождении событий.
* Улучшенное разделение ответственности.

**Недостатки CQRS**

* имеет более сложную архитектуру;
* отставание репликации.