

Ранее мы рассматривали паттерны предоставления внешнего API - Api Gateway и BFF.

**Рассмотрим подробнее, какими паттернами можно предоставлять сразу много данных для Frontend**

Допустим, у нас есть онлайн-платформа для просмотра фильмов. На этой платформе есть микросервисы для управления каталогом фильмов, управления подписками, и управления пользователями.

**API Composition**

**Что это за паттерн?**

API Composition — это паттерн, в котором один сервис (обычно BFF или Api Gateway) по синхронной интеграции собирает данные из нескольких других бэкенд сервисов, чтобы выполнить какую-то задачу или сформировать единый ответ для клиента.

**Как это реализуется в монолите?**

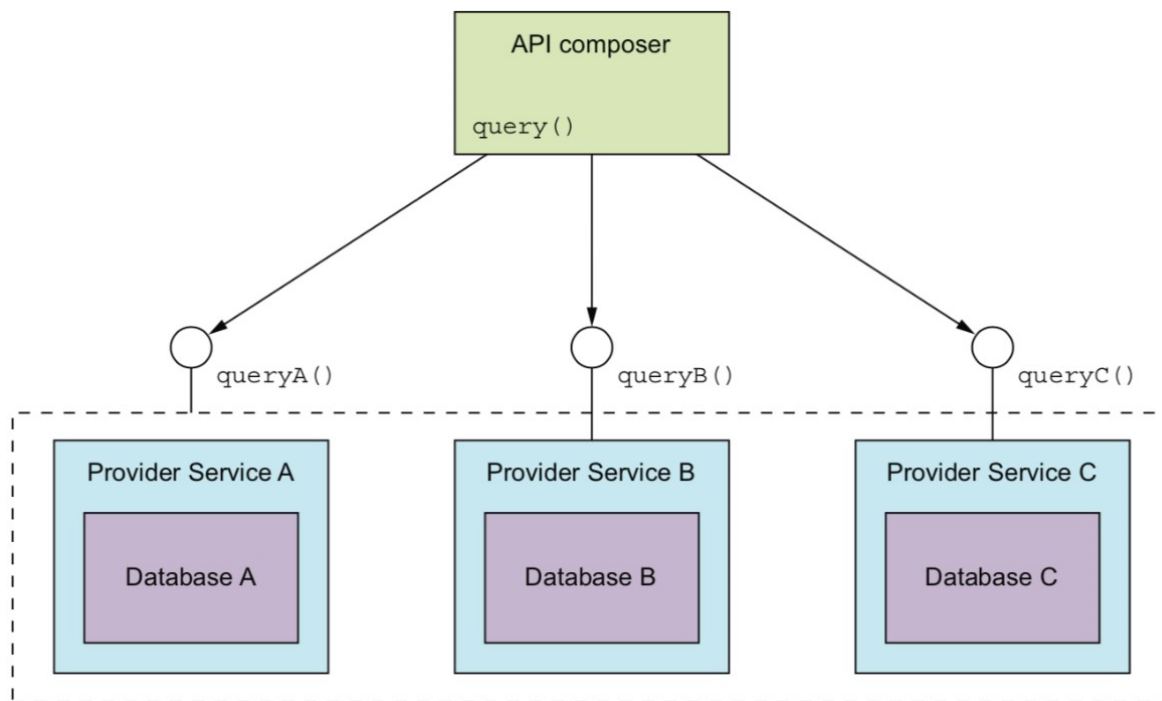
В монолитной архитектуре это может быть просто метод или функция, которая собирает данные из разных частей системы (например, из разных таблиц базы данных) и объединяет их в единый ответ.

**Зачем этот паттерн в микросервисах?**

В микросервисной архитектуре это становится необходимым, потому что данные и функциональность разбросаны по разным сервисам, и для предоставления полной картинки необходимо собрать эти части вместе.

**Как это реализуется?**

Клиент хочет увидеть список фильмов, на которые у него есть подписка, как пользователя. Сервис-агрегатор делает последовательные или параллельные синхронные запросы по API к сервису каталога фильмов, сервису подписок и сервису пользователей, объединяет полученные данные и отправляет их обратно на Frontend клиенту.



### Нюансы при использовании паттерна:

- Требуется хорошо продуманная обработка ошибок и таймаутов, так как один недоступный бэкенд сервис может "сломать" весь процесс.
- Увеличивается сетевая нагрузка из-за множества синхронных вызовов.

## CQRS (Command Query Responsibility Segregation)

### Что это за паттерн?

CQRS — это архитектурный паттерн, который разделяет функции чтения данных и функцию записи данных. В основном это применяется для улучшения производительности и масштабируемости системы.

**Как это реализуется в монолите?**

В монолите можно создать отдельные модели для чтения и записи данных, каждая со своим набором операций и, возможно, даже собственной оптимизированной базой данных для чтения. Однако это не имеет особого смысла.

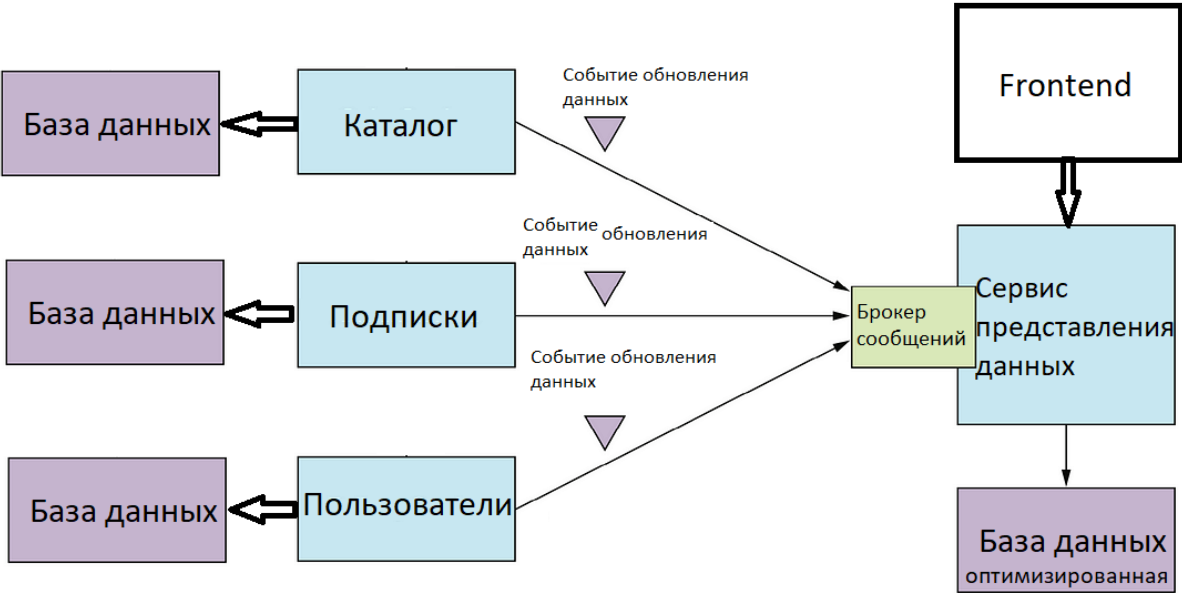
**Зачем этот паттерн в микросервисах?**

В микросервисах CQRS позволяет оптимизировать каждый микросервис для выполнения определенного типа операций (чтение или запись), что упрощает масштабирование и улучшает производительность. Это жизненно важно при высоких нагрузках сервисов типа Netflix.

**Как это реализуется?**

Сервисы каталога фильма, подписок и пользователей используются для записи данных. То есть там заложена только бизнес-логика для изменения каких-то данных. Не производится никаких операций чтения из этих сервисов.

И выделяется отдельный сервис, например "Сервис представления данных" - и он отвечает только за отображение данных из сервисов каталога фильма, подписок и пользователей. Именно с этим сервисом связан Frontend. Вся информацию для чтения он асинхронно через брокеры сообщений получает от трёх других сервисов соответственно (и складывает в свою отдельную базу данных, которая хранит только часть нужной информации из этих сервисов и оптимизирована именно для быстрого чтения информации для Frontend).



## Нюансы при использовании паттерна:

- Сложность синхронизации между сервисами чтения и записи.
- Необходимость в двух разных базах данных (для записи и для чтения) может усложнить систему.

## В итоге, что может повлиять на ваш выбор между двумя подходами:

1. **Синхронность vs Асинхронность:** В API Composition все обычно происходит синхронно, в то время как в CQRS обновление модели чтения может происходить асинхронно.
2. **Сложность:** CQRS обычно более сложен для реализации из-за необходимости управлять двумя разными моделями (чтение и запись).
3. **Производительность и масштабируемость:** CQRS может предложить лучшую производительность и масштабируемость, а также возможность реализации разных сложных запросов в микросервисной архитектуре.
4. **Использование ресурсов:** API Composition может потребовать больше сетевых ресурсов, потому что ему нужно сделать много синхронных вызовов к другим сервисам.