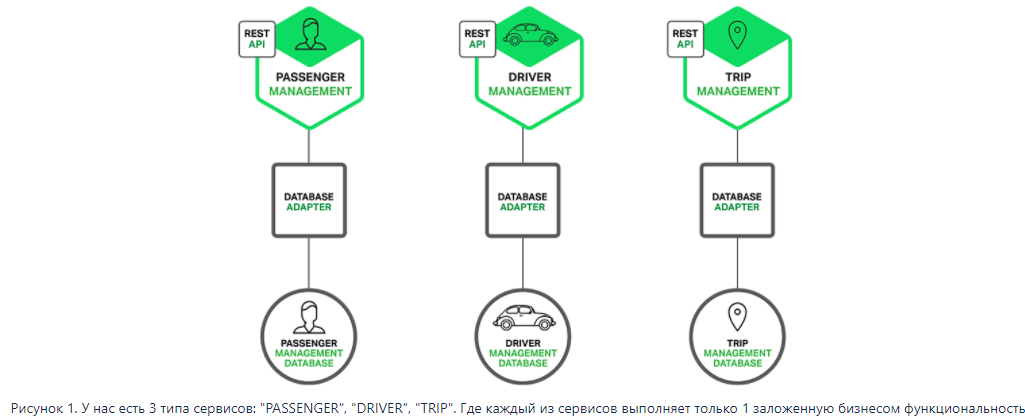
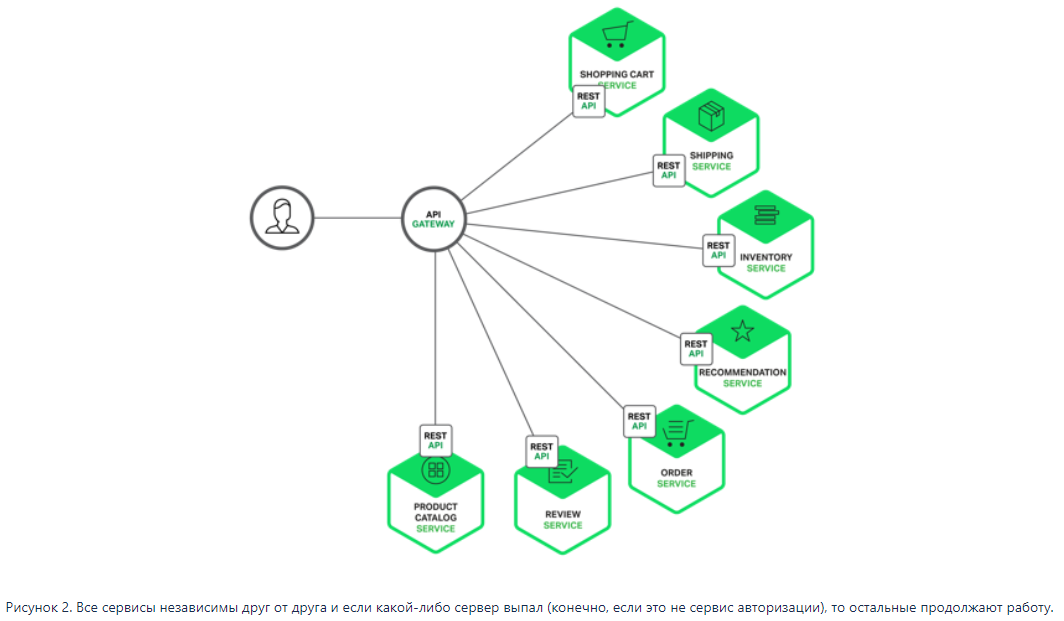
**Микросервис** - это архитектурный стиль, который структурирует приложение как набор небольших автономных сервисов, смоделированных вокруг бизнес-сферы. Большое приложение можно разделить на несколько небольших служб. Каждый сервис выполняет свою соответствующую задачу, чтобы обеспечить надлежащее функционирование всей системы. Эти услуги независимы друг от друга. Поэтому микросервисы облегчают процесс разработки.

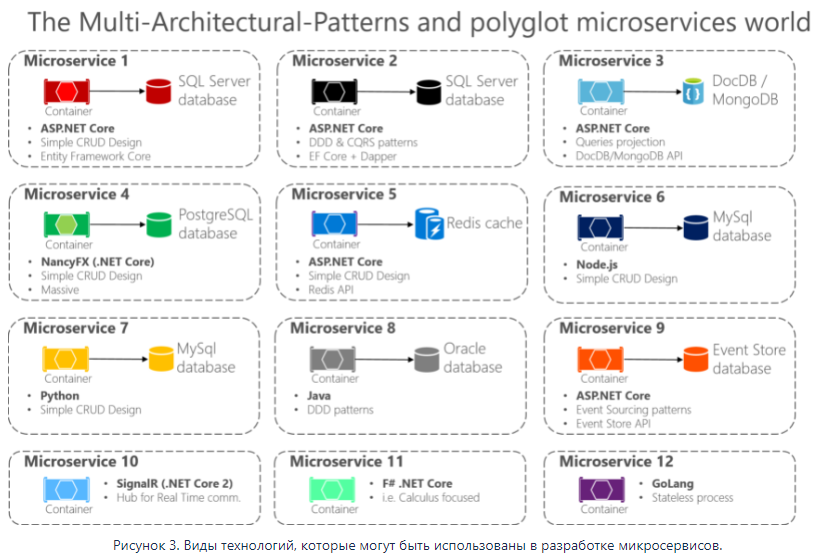
* **Небольшие** - под этим понятием мы ожидаем, что наш микросервис будет работать только с одной частью функциональности для которой он разрабатывался в рамках бизнеса. Говорят, что микросервис можно оценить по кол-ву строк кода и если он превышает какое-то кол-во этих строк, то микросервис перестаёт быть таковым. И тут стоит вопрос. Как мы узнаем, микросервис это или нет? Мы знаем точно, что за одним микросервисом обычно стоит небольшая команда разработчиков, а иногда - это и вовсе один разработчик, и если он может легко поддерживать, и понимать всю бизнес-логику сервиса один, то гуд, а если нет, то время его пилить микросервис на несколько частей.



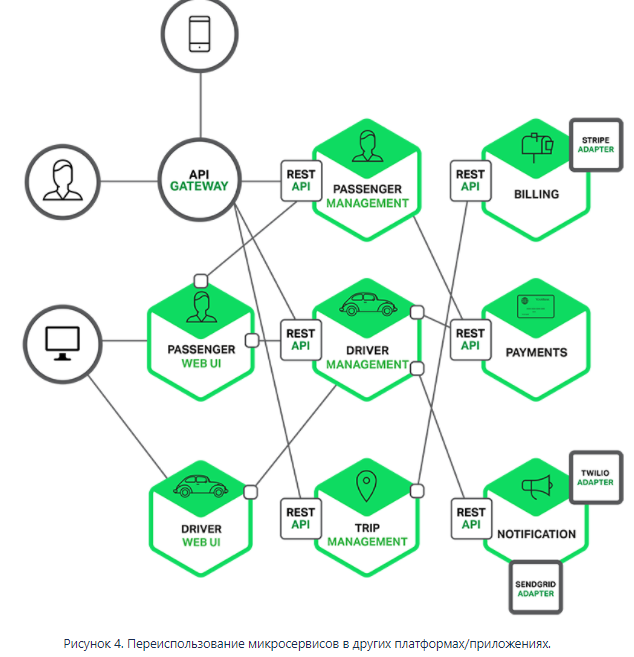
* **Независимые и автономные** - микросервисы изолированы друг от друга и их стараются создавать на отдельных "машинах". Изменения на этих сервисах проходят независимо от других сервисов и и деплои апдейтов, как правило, не влияют на того, кто ранее его использовал.



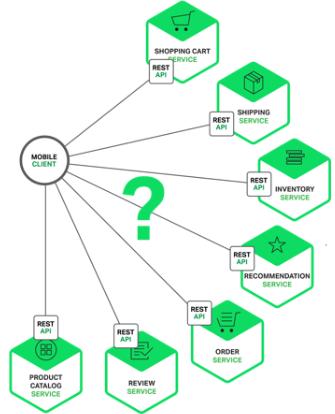
* **Устойчивы к сбоям** - это вытекает из прошлого пункта. И если у нас откажет один сервис, другие продолжат свою работу.
* **Масштабируемость** - ещё одна фишка микросервисов. Когда растёт нагрузка на какой-то сервис, мы можем отдельно выделить больше ресурсов e.g. CPU, RAM, HDD/SDD хранилища и др. на него. т.к. все наши сервисы находятся отдельно друг от друга в отличии от монолита, где мы не можем так легко распределять эти ресурсы.
* **Лёгкость деплоя** - деплой внутри одного сервиса уменьшает риски того, что вся система ляжет. Сам деплой одного микросервиса будет в разы быстрее, чем целой монолитной структуры. И если что-то отломалось, вернуть прошлое состояние будет гораздо проще. Выкатить новую фичу в лайв теперь можно гораздо быстрее и безопаснее.
* **Написаны на разных языках (используют различные технологии)** - такой подход сильно упрощает гибкость в разработке проекта. В отличии монолита, вы не привязаны к использованию одного языка программирования или использованию определённого типа базы данных на проекте. Таким способом, использую различные технологии, можно добиться повышения производительности работы вашего проекта в целом. Переход от старой технологии, к новой, происходит менее болезненно, ведь переписывается только один сервис. + Есть возможность независимо проверить, как отработает новая технология в целой системе.



* **Переиспользование сервисов** - микросервисы могут быть легко переиспользованы в других приложения на других платформах или проектах. e.g. Мобильные приложения, web приложения, standalone приложения



**Взаимодействие микросервисов (Клиент > Сервис)**



**У нас есть несколько вариантов как мы будем выдавать информацию с этих сервисов клиенту:**

* Direct Client‑to‑Microservice Communication: Стучимся в микросервис и получаем информацию напрямую.
* Используя API Gateway

**Direct Client‑to‑Microservice Communication**

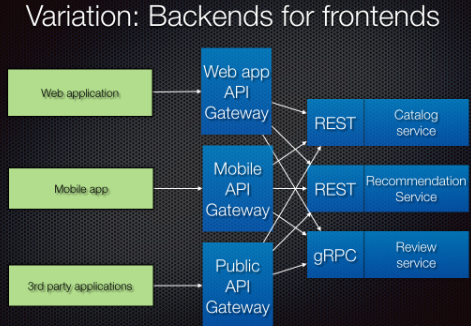
Приложение на прямую стучит в определённый **эндпоинт**, этот URL подвязан на **балансер**, который запрашивает данные о продукте. Мобильный клиент делает запросы во все 7 микросервисов на наличие данных и возвращает их. На деле, это выливается в то, что сам клиент становится очень большой. Что для мобильных приложений не является хорошим знаком. И если при небольшом кол-ве сервисов этот вариант смотрится не плохо, то представим, что у нас тысячи микросервисов.

Второй вариант почему плохо использовать клиент напрямую для вытягивания данных - это протоколы, которые используют наши веб-сервисы. Рассмотрим, что один сервис использует **RPC** **(Remote Procedure Call)** протокол, а другой, в свою очередь **AMQP** **(Advanced Message Queuing Protocol)**. Оба этих протокола не особо дружелюбны к файрволу и браузеру. По этому для приложений обычно используют **HTTP + WebSocket**.

И последний пункт, почему он не является хорошим выбором - это сложность в рефакторинге микросервисов. А именно в том, когда мы захотим объединенить два сервиса в один или разбить микросервис на несколько мы скорее всего получим только головную боль.

**API Gateway**

Дабы избежать проблем описаных выше, используют **API Gateway**. Это сервис, который хранит данные сервисов и преобразует их для каждого клиента в том, формате, который необходим. На примере ниже, мы можем увидеть, как работает API Gateway для WEB приложения, мобильного приложения и 3rd party. По сути своей он является переводчиком-проводником и не должен содержать никакой сложной бизнес логики.



**Есть 3 основных способа, по которым сервисы общаются (как API Gateway находит микросервисы):**

1. Service Discovery (RPC Style)
   1. Server-Side Service Discovery
   2. Client-Side Service Discovery
2. Message Bus (Event-driven)
3. Hybrid

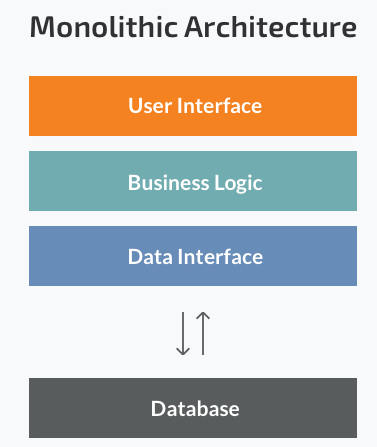
**Monolithic vs Microservices**

Having come into light just a few years ago, microservices are an accelerating trend these days. Indeed, microservices approach offers tangible benefits including an increase in scalability, flexibility, agility, and other significant advantages. Netflix, Google, Amazon, and other tech leaders have successfully switched from monolithic architecture to microservices. Meanwhile, many companies consider following this example as the most efficient way for business growth.

On the contrary, the monolithic approach is a default model for creating a software application. Still, its trend is going down because building a monolithic application poses a number of challenges associated with handling a huge code base, adopting a new technology, scaling, deployment, implementing new changes and others.

**The monolithic architecture** is considered to be a traditional way of building applications. A monolithic application is built as a single and indivisible unit. Usually, such a solution comprises a client-side user interface, a server side-application, and a database. It is unified and all the functions are managed and served in one place.

Normally, monolithic applications have one large code base and lack modularity. If developers want to update or change something, they access the same code base. So, they make changes in the whole stack at once.



**Strengths of the Monolithic Architecture**

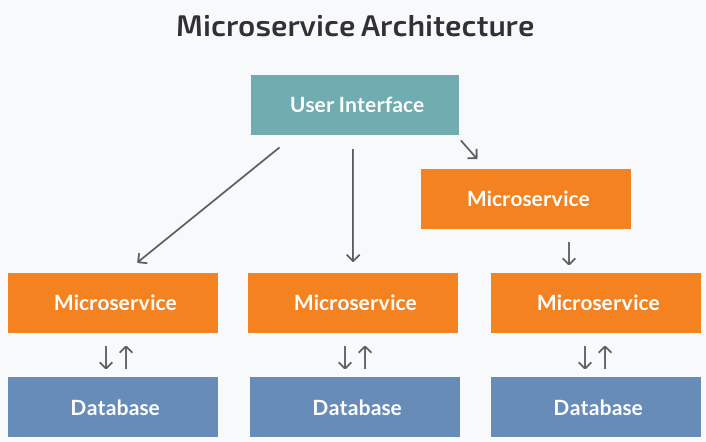
* **Less cross-cutting concerns**. Cross-cutting concerns are the concerns that affect the whole application such as logging, handling, caching, and performance monitoring. In a monolithic application, this area of functionality concerns only one application so it is easier to handle it.
* **Easier debugging and testing**. In contrast to the microservices architecture, monolithic applications are much easier to debug and test. Since a monolithic app is a single indivisible unit, you can run end-to-end testing much faster.
* **Simple to deploy**. Another advantage associated with the simplicity of monolithic apps is easier deployment. When it comes to monolithic applications, you do not have to handle many deployments – just one file or directory.
* **Simple to develop**. As long as the monolithic approach is a standard way of building applications, any engineering team has the right knowledge and capabilities to develop a monolithic application.

**Weaknesses of the Monolithic Architecture**

* **Understanding**. When a monolithic application scales up, it becomes too complicated to understand. Also, a complex system of code within one application is hard to manage.
* **Making changes**. It is harder to implement changes in such a large and complex application with highly tight coupling. Any code change affects the whole system so it has to be thoroughly coordinated. This makes the overall development process much longer.
* **Scalability**. You cannot scale components independently, only the whole application.
* **New technology barriers**. It is extremely problematic to apply a new technology in a monolithic application because then the entire application has to be rewritten.

While a monolithic application is a single unified unit, **a** **microservices** **architecture** breaks it down into a collection of smaller independent units. These units carry out every application process as a separate service. So all the services have their own logic and the database as well as perform the specific functions.

Within a microservices architecture, the entire functionality is split up into independently deployable modules which communicate with each other through defined methods called APIs (Application Programming Interfaces). Each service covers its own scope and can be updated, deployed, and scaled independently.



**Strengths of the Microservice Architecture**

* **Independent components**. Firstly, all the services can be deployed and updated independently, which gives more flexibility. Secondly, a bug in one microservice has an impact only on a particular service and does not influence the entire application. Also, it is much easier to add new features to a microservice application than a monolithic one.
* **Easier understanding**. Split up into smaller and simpler components, a microservice application is easier to understand and manage. You just concentrate on a specific service that is related to a business goal you have.
* **Better scalability**. Another advantage of the microservices approach is that each element can be scaled independently. So the entire process is more cost- and time-effective than with monoliths when the whole application has to be scaled even if there is no need in it. In addition, every monolith has limits in terms of scalability, so the more users you acquire, the more problems you have with your monolith. Therefore, many companies, end up rebuilding their monolithic architectures.
* **Flexibility in choosing the technology**. The engineering teams are not limited by the technology chosen from the start. They are free to apply various technologies and frameworks for each microservice.
* **The higher level of agility**. Any fault in a microservices application affects only a particular service and not the whole solution. So all the changes and experiments are implemented with lower risks and fewer errors.

**Weaknesses of the Microservice Architecture**

* **Extra complexity**. Since a microservices architecture is a distributed system, you have to choose and set up the connections between all the modules and databases. Also, as long as such an application includes independent services, all of them have to be deployed independently.
* **System distribution**. A microservices architecture is a complex system of multiple modules and databases so all the connections have to be handled carefully.
* **Cross-cutting concerns**. When creating a microservices application, you will have to deal with a number of cross-cutting concerns. They include externalized configuration, logging, metrics, health checks, and others.
* **Testing**. A multitude of independently deployable components makes testing a microservices-based solution much harder.

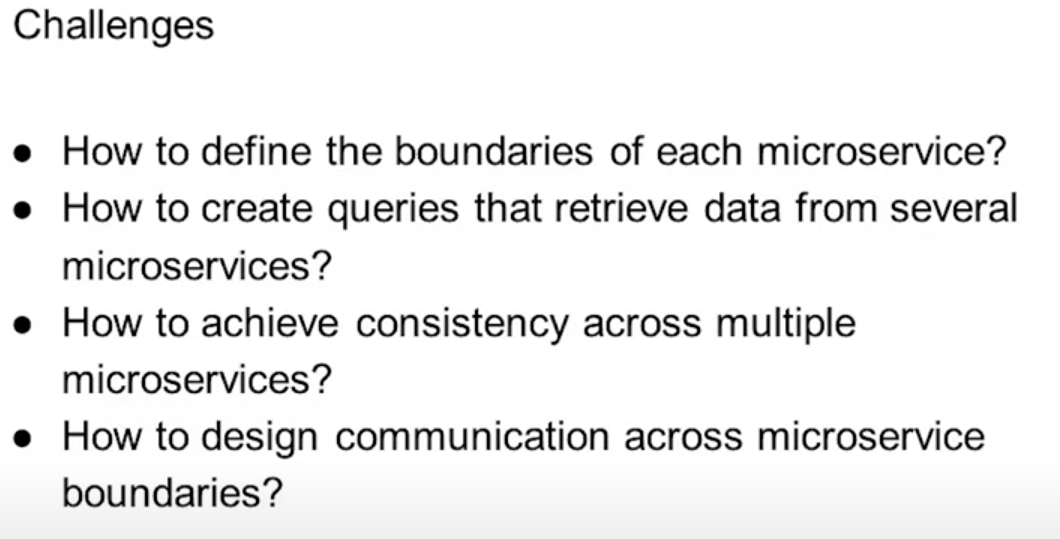
**WHICH SOFTWARE ARCHITECTURE SUITS YOUR SOLUTION AND YOUR BUSINESS BEST?**

**Choosing a monolithic architecture**

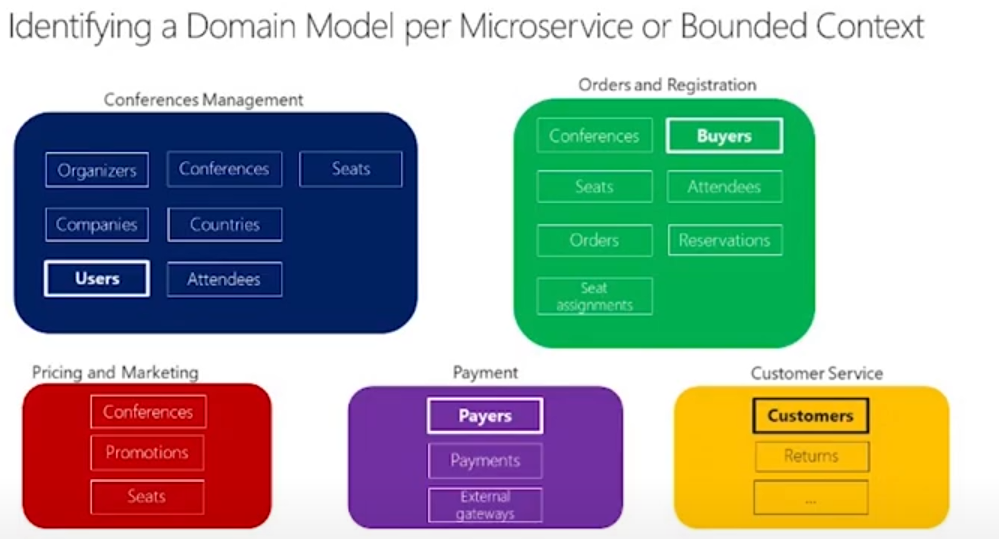
* **Small team**. If you are a startup and your team is small, you may not need to deal with the complexity of the microservices architecture. A monolith can meet all your business needs so there is no emergency to follow the hype and start with microservices.
* **A simple application**. Small applications which do not demand much business logic, superior scalability, and flexibility work better with monolithic architectures.
* **No microservices expertise**. Microservices require profound expertise to work well and bring business value. If you want to start a microservices application from scratch with no technical expertise in it, most probably, it will not pay off.
* **Quick launch**. If you want to develop your application and launch it as soon as possible, a monolithic model is the best choice. It works well when you aim to spend less initially and validate your business idea.

**Choosing a microservices architecture**

* **Microservices expertise**. Without proper skills and knowledge, building a microservice application is extremely risky. Still, just having the architecture knowledge is not enough. You need to have DevOps and Containers experts since the concepts are tightly coupled with microservices. Also, domain modelling expertise is a must. Dealing with microservices means splitting the system into separate functionalities and dividing responsibilities.
* **A complex and scalable application**. The microservices architecture will make scaling and adding new capabilities to your application much easier. So if you plan to develop a large application with multiple modules and user journeys, a microservice pattern would be the best way to handle it.
* **Enough engineering skills**. Since a microservice project comprises multiple teams responsible for multiple services, you need to have enough resources to handle all the processes.

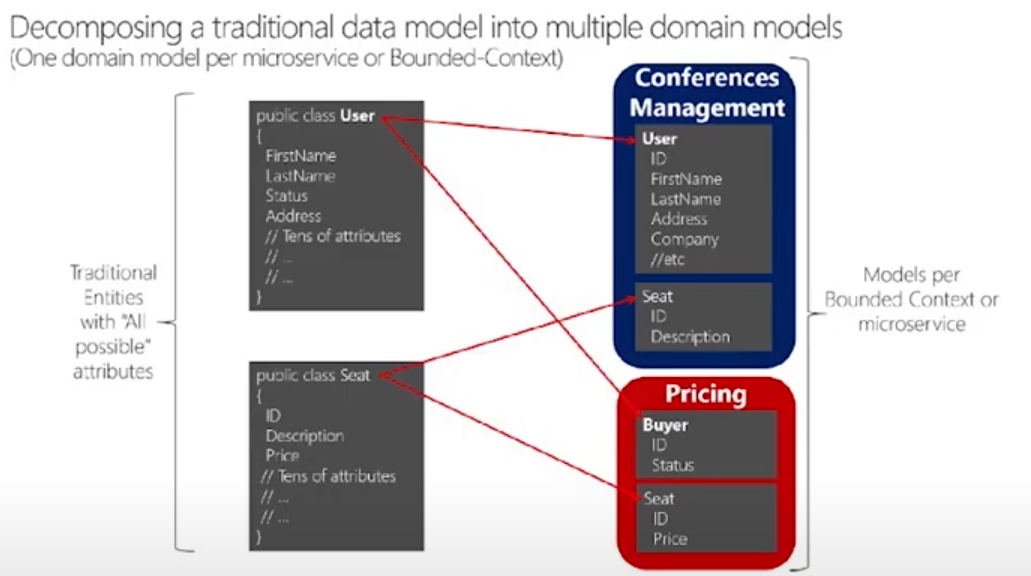
****

**How to define the boundaries of each microservice**

****

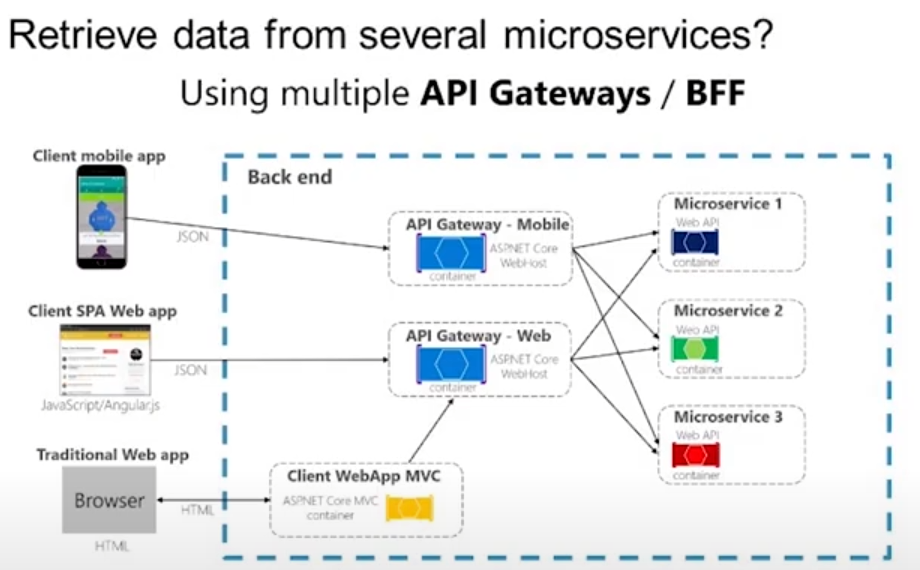
По сути мы можем сделать что бы каждый микросервис включал какую то сущность. Например, микросервис для Юзеров, Микросервис для постов, Микросервис комментов к постам. Все это замечательно, до тех пор, пока мы не решим собрать инфу по юзерам, их постам и коментам. В таком случае каждый микросерис будет делать кучу запросов, и превратится в Chatty API. И все преимущество микросервисов будет стерто

Из за этого каждый микросервис должен включать часть другого сервиса, что бы ему не нужно было к нему обращаться каждый раз. Данные могут быть не 100% синхронезированы, но зато так увеличивается быстродействие. Дублирование данных в микросервисной архитектуре – не проблема. НО у каждого микросервиса все равно должна быть своя БД, где он все сохраняет

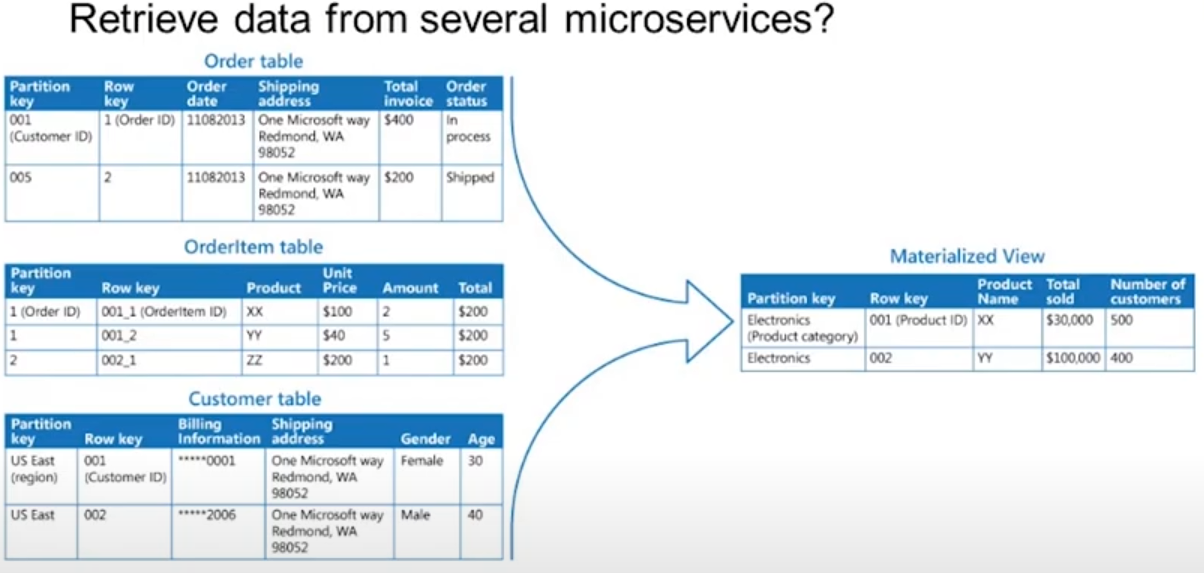


**How to create queries that retrieve data from multiple services**

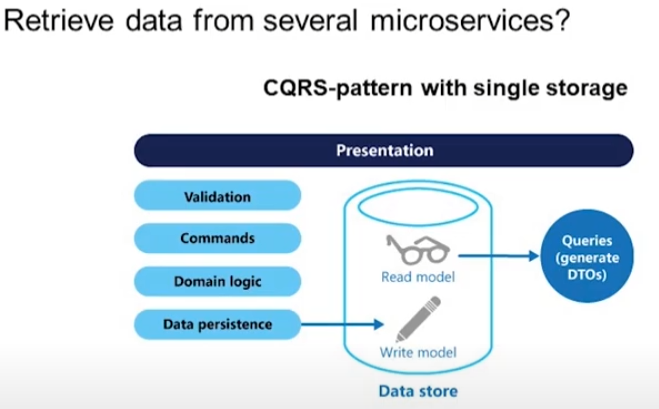
**Первый подход** – добавление API gateway

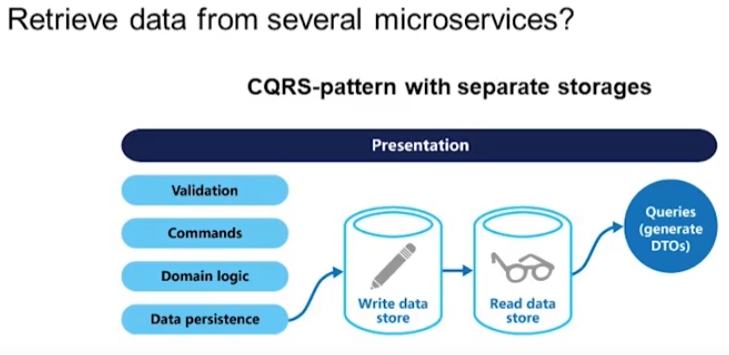


**Второй подход** – создание вьюшек в БД

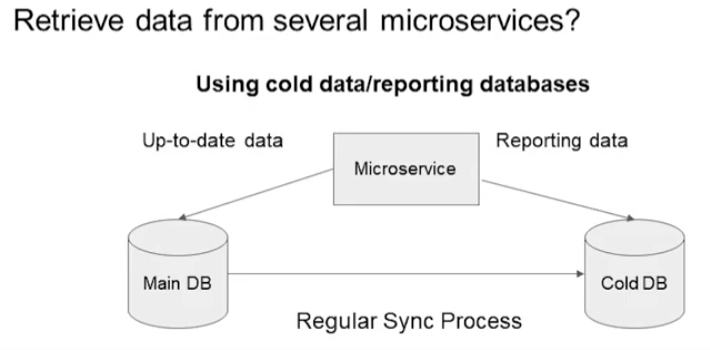


**Третий подход**



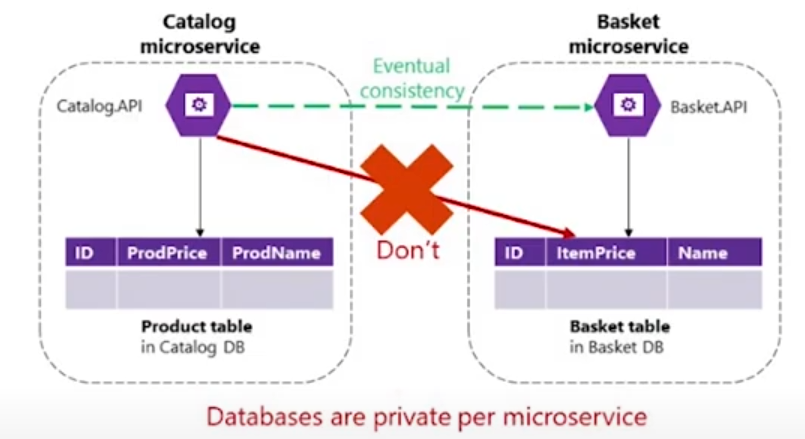


**Четвертый подход**

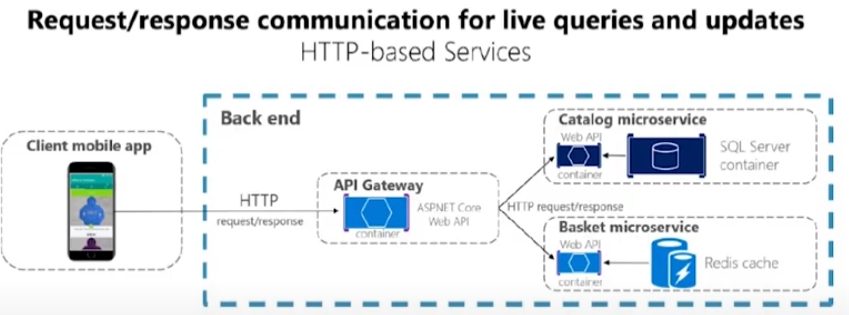


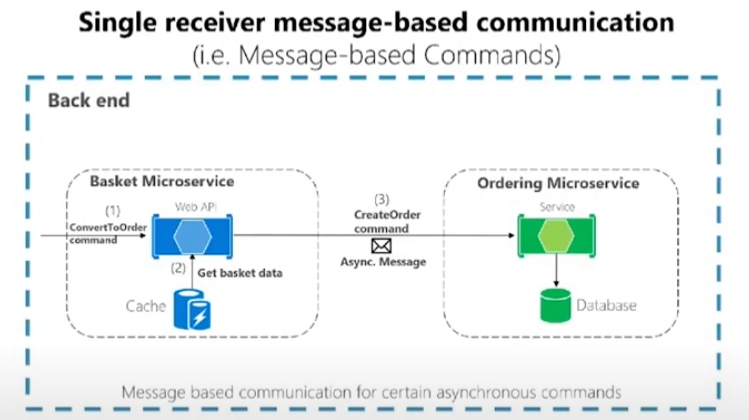
**How to achieve consistency across multiple mictoservices**

Нельзя что бы один микросервис писал данные в другой микросервис

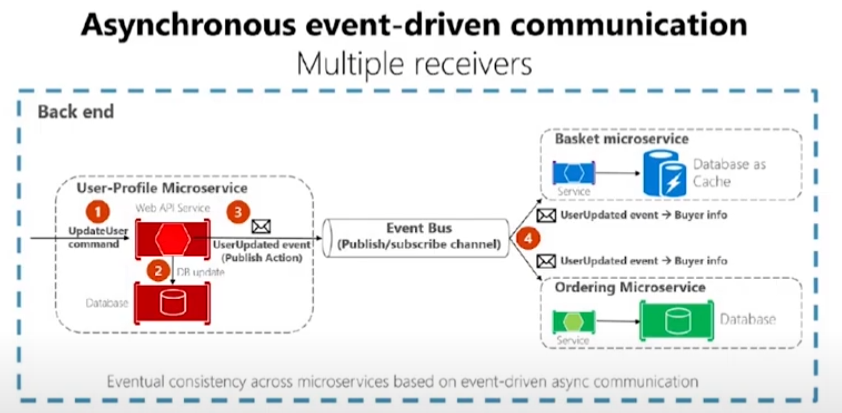


**Использовать REST API**





**Использовать брокеры сообщений (Kafka, RabbitMQ)**



**Синхронную коммуникацию лучше не использовать**

