Microservices

Lev Epshtein
lev@opsguru.io

NAYA College

About me

Lev Epshtein

Technology enthusiast with 15 years of industry experience in DevOps and IT. Industry experience with back-end architecture.

Solutions architect with experience in big scale systems hosted on AWS/GCP, end-to-end DevOps automation process.

Cloud DevOps, and Big Data instructor at NAYA and JB.

Certified GCP trainer,

Partner & Solution Architect Consultant at Opsguru.

lev@opsguru.io

NAYA College

MSA

Microservices architecture

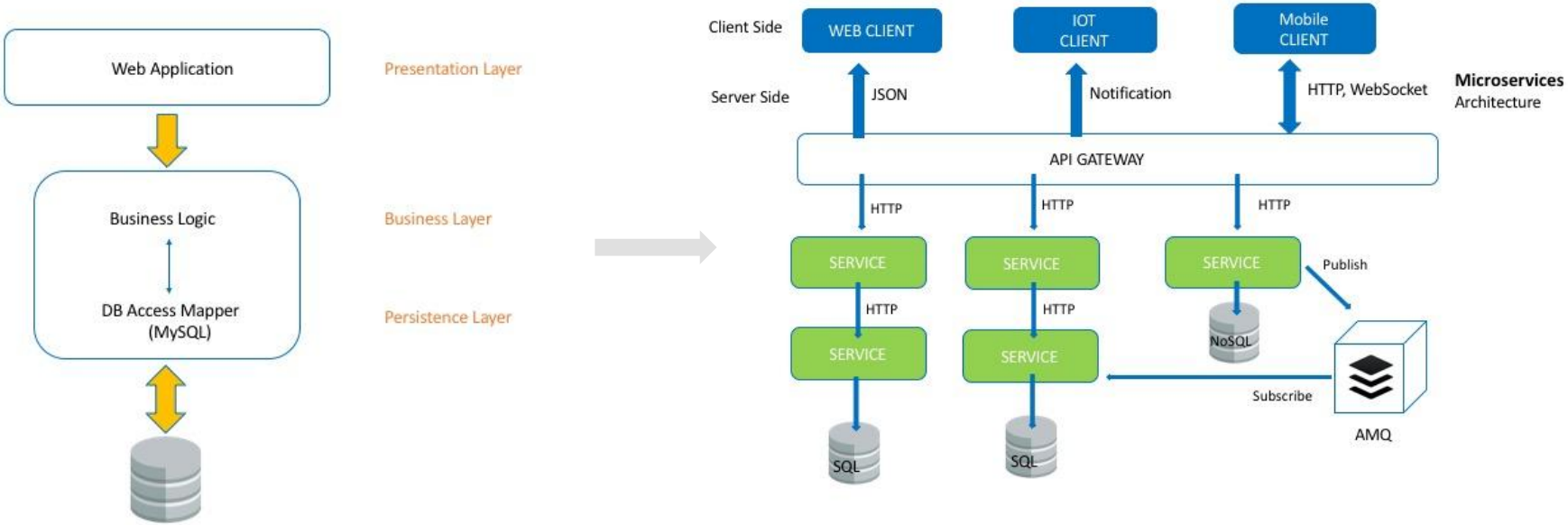
- Monolith architecture vs MSA
- Monolith architecture to MSA

NAYA College

Monolith architecture vs MSA

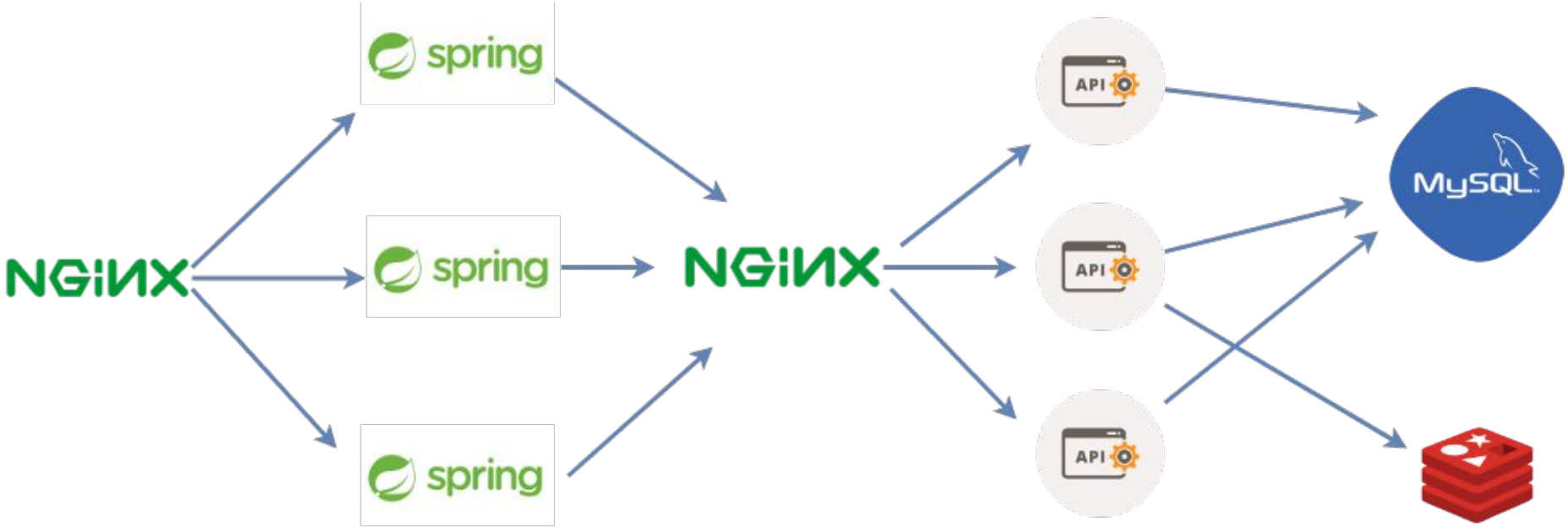
What is A Microservices Architecture

Microservice architecture is the “latest” (since 2014…) method of developing software as a suite of small modular and independently deployable applications.



Small decoupled services / API Layer

No shared libraries / common code



NAYA College

CONS Monolithic Application

- A nightmare to manage in large scale due to the fact that developers keeps on adding new features and changes which makes it complex and difficult to fully understand what does what.
- Everything is written and coded in the same programing language.
- Each deploy -> full deploy to the whole system
- Reliability and Bug hunting (Full regression anyone?)
- Change / Upgrade framework - Easier to land on the moon (again)

Microservices Benefits

- Application complexity by decomposing an application into a set of manageable services which are faster to develop and much easier to understand and maintain.
- Splitting a big application in a set of smaller services also improve the fault isolation. In fact, it is more difficult for the whole system to down at the same time and a failure in processing one customer’s request is less likely to affect other customer’s requests.
- Developers are free to choose whatever technologies make sense for their service. We are not more bound to the technologies chosen at the start of the project - Wooo Hooo!
- Easier for a new developer to understand the reduced set of functionality of a microservice instead of understand a big application design

Microservices Benefits continue

- Each microservice could be refactored piece by piece as new technology solutions become available
- Language agnostic, so we can select the appropriate language or framework for each service
- Each service can be deployed independently by a team that is focused on that service, using different technology stacks that are best suited for their purposes. This is great for continuous delivery, allowing frequent releases while keeping the rest of the system stable.

Microservices Benefits continue

- Continuous deployment possible for complex applications and enables each service to be scaled independently.
- Eliminate vendor or technology lock-in: Microservices provide the flexibility to try out a new technology stack on an individual service as needed. There won't be as many dependency concerns and rolling back changes becomes much easier. With less code in play, there is more flexibility.
- Scalability: Since your services are separate, you can more easily scale the most needed ones at the appropriate times, as opposed to the whole application. When done correctly, this can impact cost savings.

Microservices Disadvantages

- **Communication** between services is complex: Since everything is now an independent service, you have to carefully handle requests traveling between your modules. In one such scenario, developers may be forced to write extra code to avoid disruption. Over time, complications will arise when remote calls experience latency.
- **More services equals more resources**: Multiple databases and transaction management can be painful.
- **Global testing is difficult**: Testing a microservices-based application can be cumbersome. In a monolithic approach, we would just need to launch our WAR on an application server and ensure its connectivity with the underlying database. With microservices, each dependent service needs to be confirmed before testing can occur.

Microservices Disadvantages

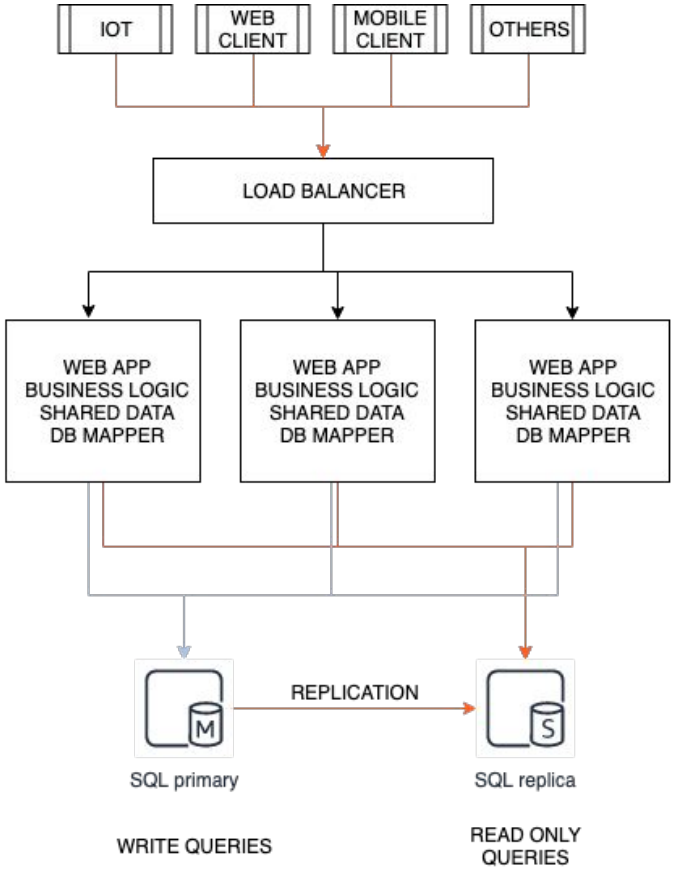
- **Debugging problems can be harder**: Each service has its own set of logs to go through. Log, logs, and more logs.
- **Deployment challengers**: The product may need coordination among multiple services, which may not be as straightforward as deploying a WAR in a container.
- **Large vs small product companies**: Microservices are great for large companies, but can be slower to implement and too complicated for small companies who need to create and iterate quickly, and don’t want to get bogged down in complex orchestration.

Monolith VS Microservices: Scaling

NAYA College

Monolith Scaling

Monolith apps requires to scale the entire monolith application to achieve scalability
But what if one component requires much more resource than others?

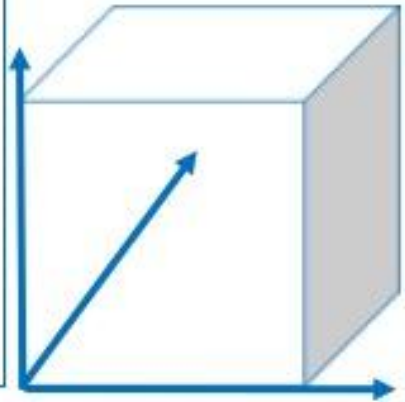


NAYA College

Microservices: Scaling

Y axis

Functional Decomposition
Scale by splitting application into **multiple**, **different** services.
Each service is responsible **for one or more closely related functions**



Z axis

Data Partitioning
Scale by running multiple copies of an application.
Similar to X-axis scaling but each server is responsible for **only a subset of the data**

X axis

Horizontal Scale
Scale by running multiple copies of an application behind a load balancer

Communication

Communication

When we start thinking about MSA we will imagine the following use case:

- A system with 2 Servers / Services A and B that need to communicate with each other

First thing that comes into our mind is

- **REST API**

Communication

REST API ARE GREAT! FOR MICROSERVICES

A microservice API is a contract between the service and its clients. You'll be able to evolve a microservice independently BUT only if you do not break its API contract, which is why the contract is so important. If you change the contract, it will impact your client applications or your API Gateway.

Communication

However, even we were thoughtful about our initial contract, a service API will need to change over time. When that happens—and especially if our API is a public API consumed by multiple client applications — we typically can't force all clients to upgrade to your new API contract.

We will have to incrementally deploy new versions of a service in a way that both old and new versions of a service contract are running simultaneously. Therefore, it's important to have a strategy for your service versioning and rollback!

Communication

BUT THEN...

What about if B becomes very slow to respond? Communication with REST API is synchronous.

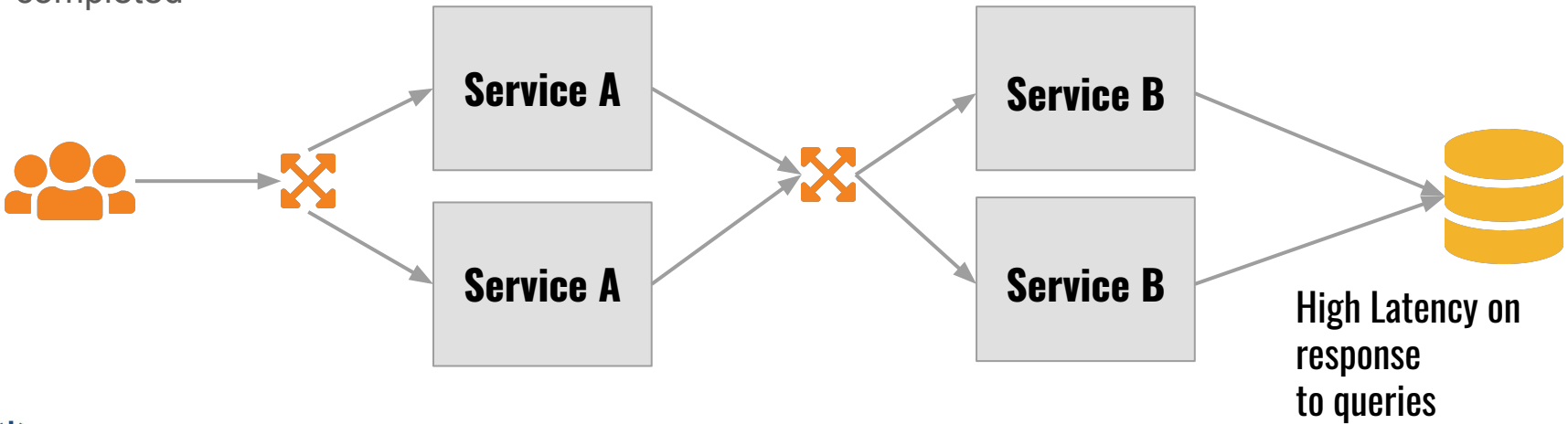
Meaning that If B is slow, then A will be slow and eventually might crash.

Think about the following scenario

- ‘A’ need to GET information from B while B pull that information from a DB

Due to high Load / Traffic -

Our Database suffer performance degradation and requests from B takes a long time to be completed



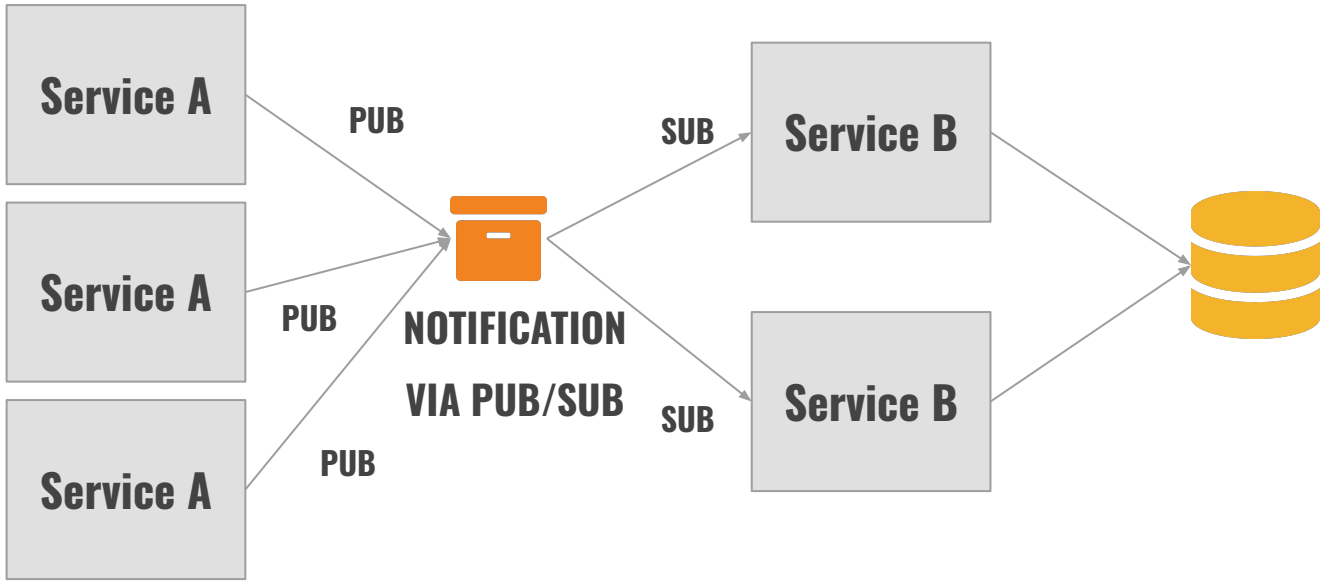
Result?

- Service A will immediately be affected and slow down and even crash due to out of resources or pool requests depleted as all requests are caught and waiting for response from Service B
- Transactions we had in our Service A Servers/Containers will be lost (if Service A will crash) without and option to recover them.
- User/Service Experience will be terrible as we will have to ask or impose some sort of retry mechanism - DON’T

THE CAUSE - SYNCHRONOUS

We can make the communication asynchronous with a pub/sub store like Kafka:

“Publish/Subscribe messaging system. allows producers to write records into X that can be read by one or more Subscribers”



NAYA College

Using PUB/SUB:

- It should provide asynchronous communication (decouple processes)
- It should be scalable to support the growth of SERVICE B and SERVICE A
- It should be distributed to guarantee high availability (HA)
- The FIFO (First In First Out) of messages guarantee or at least, a mechanism should exist to manage message order where FIFO is a must.
- No messages should be lost on receivers down time (during a short period of time)
- It should enable load balancing on receivers with 2 modes:
  - All receivers receive all and same messages (mode pub/sub)
  - Messages are distributed among receivers (mode queue)

Delivered once Consumed Once

More

- **Decoupled Architecture** — The sender's job is to send messages to the queue and the receiver's job is to receive messages from the queue. This separation allows for components to operate independently, while still communicating with each other.
- **Fault tolerant** — If the consumer server is down, the messages would still be stored in queue until the server goes back up and retrieves it
- **Scalability** —We can push all the messages in the queue and configure the rate at which the consumer server pulls from it. This way, the receiver won't get overloaded with network requests and can instead maintain a steady stream of data to process.

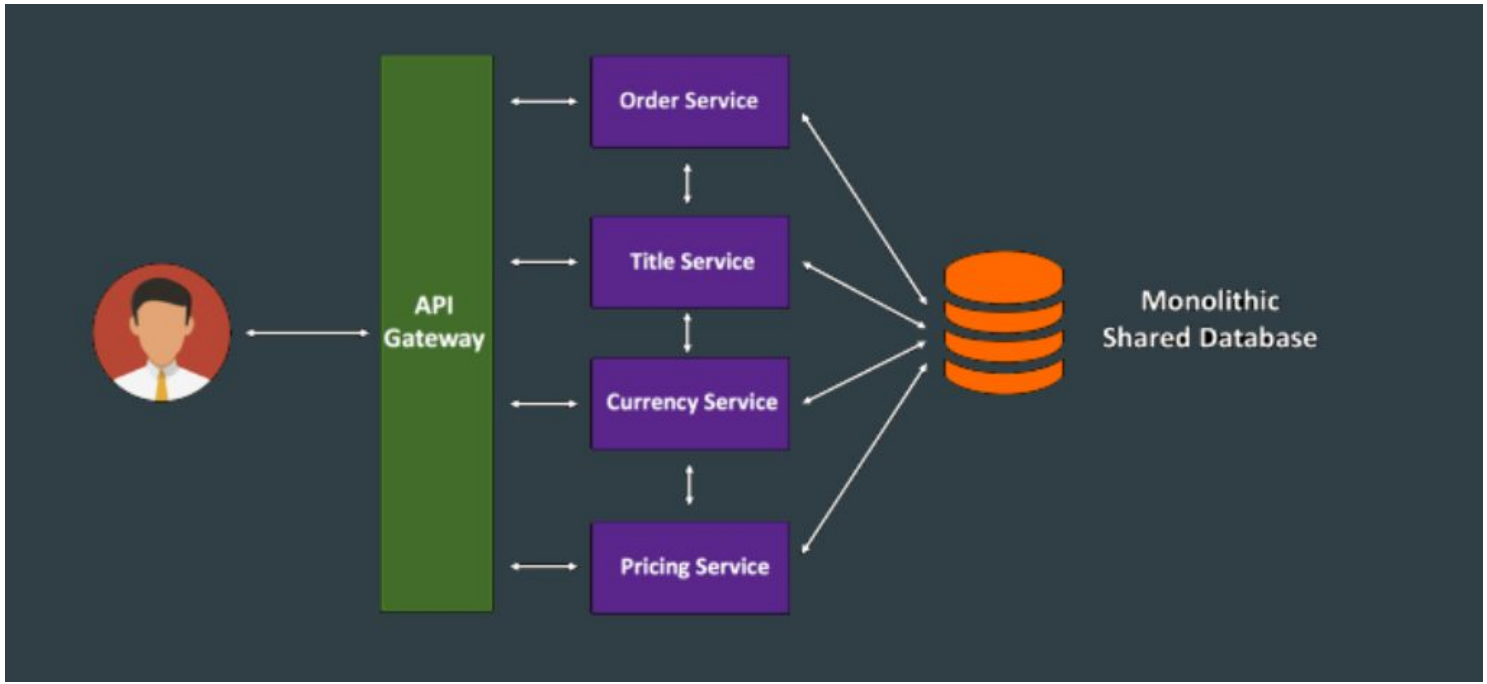
DATA

PATTERN: DATABASE PER SERVICE

One should design his microservices architecture in such a way that each individual microservice has its own separate database with its own domain data.

Traditional applications have a single shared database and data is often shared between different components resulting a monolithic design due to the old good encapsulation.

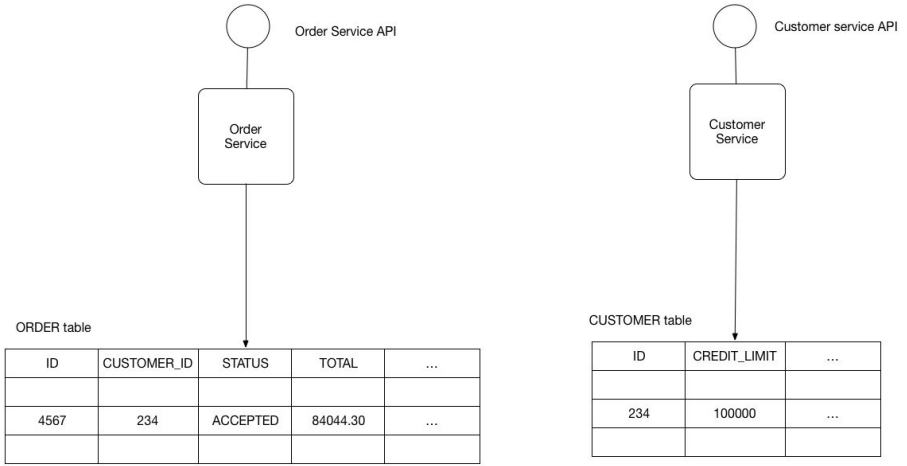
Monolith architecture - Centralized Database



NAYA College

PATTERN: DATABASE PER SERVICE

Let’s imagine an online store application using the Microservice architecture pattern. Most services need to persist data in some kind of database. For example, the Order Service stores information about orders and the Customer Service stores information about customers.



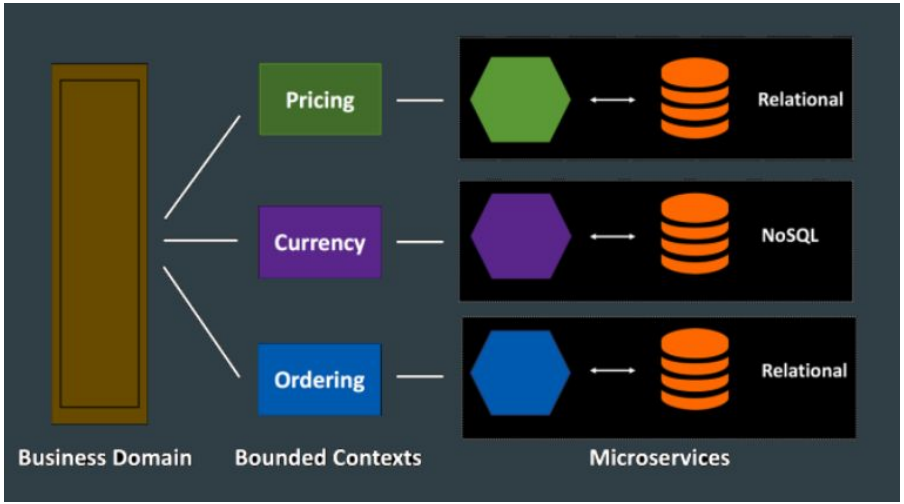
PATTERN: DATABASE PER SERVICE

This design pattern is great and allows:

Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB or maybe ES which is good at storing complex, unstructured data.

PATTERN: DATABASE PER SERVICE

While designing your database, look at the application functionality and determine if it needs a relational schema or not. Keep your mind open towards a NoSQL DB as well if it fits your criteria.



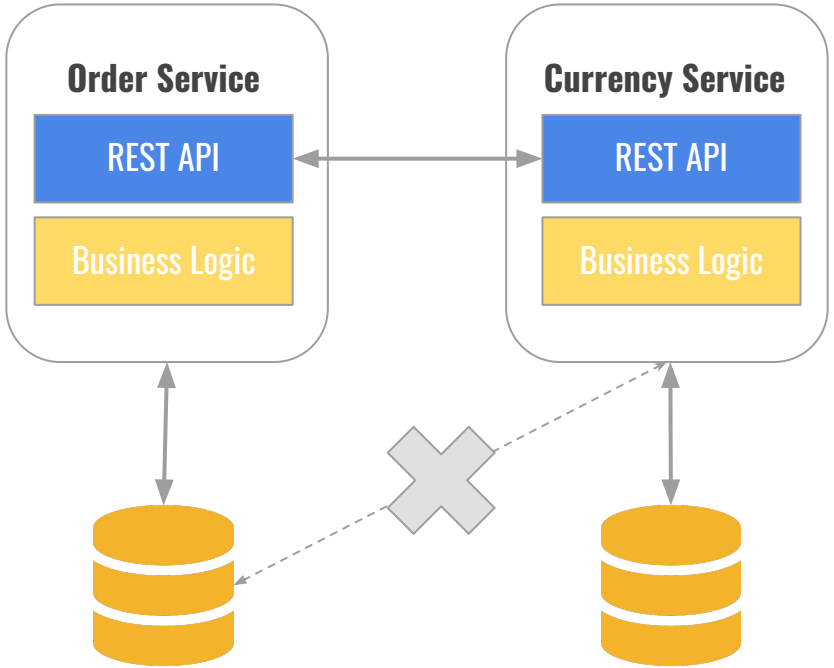
NAYA College

PATTERN: DATABASE PER SERVICE

This design pattern is great and allows:

- Services to be loosely coupled so that they can be developed, deployed and scaled independently
- Some business transactions must enforce invariants that span multiple services.
    - For example, the Place Order use case must verify that a new Order will not exceed the customer's credit limit. Other business transactions, must update data owned by multiple services.
- Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.
- Databases must sometimes be replicated and sharded in order to scale. See the Scale Cube.

Keep each microservice persistent data private to that service and accessible only via its API.



DECENTRALIZED DATA MANAGEMENT

There are a few different ways to keep a service's persistent data private.

We do not need to provision a database server for each service.

For example:

- Private-tables-per-service – each service owns a set of tables that must only be accessed by that service
- Schema-per-service – each service has a database schema that's private to that service
- Database-server-per-service – each service has it's own database server.

Monolith to MSA

Refactoring a monolith to microservices

Truly greenfield development of microservices-based applications is relatively rare. Many organizations that want to adopt microservices already have a monolithic application. The recommended approach is to use the **Strangler application pattern** and incrementally migrate function from the monolith into services.

Refactoring strategies

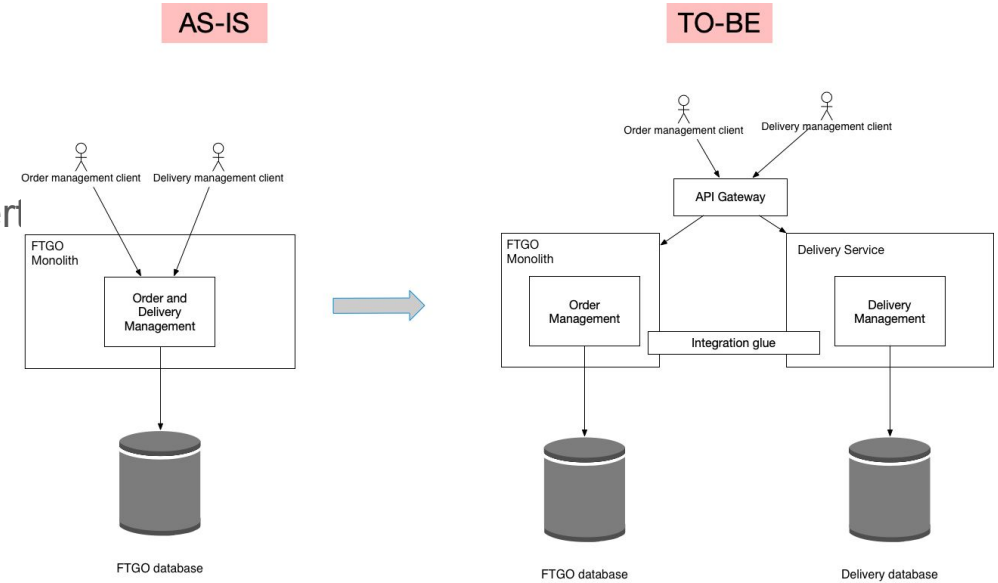
- Implement new functionality as services
- Extract services from the monolith

Implement new functionality as services

A good way to begin the migration to microservices is to implement significant new functionality as services. This is sometimes easier than breaking apart of the monolith. It also demonstrates to the business that using microservices significantly accelerates software delivery.

Extract services from the monolith

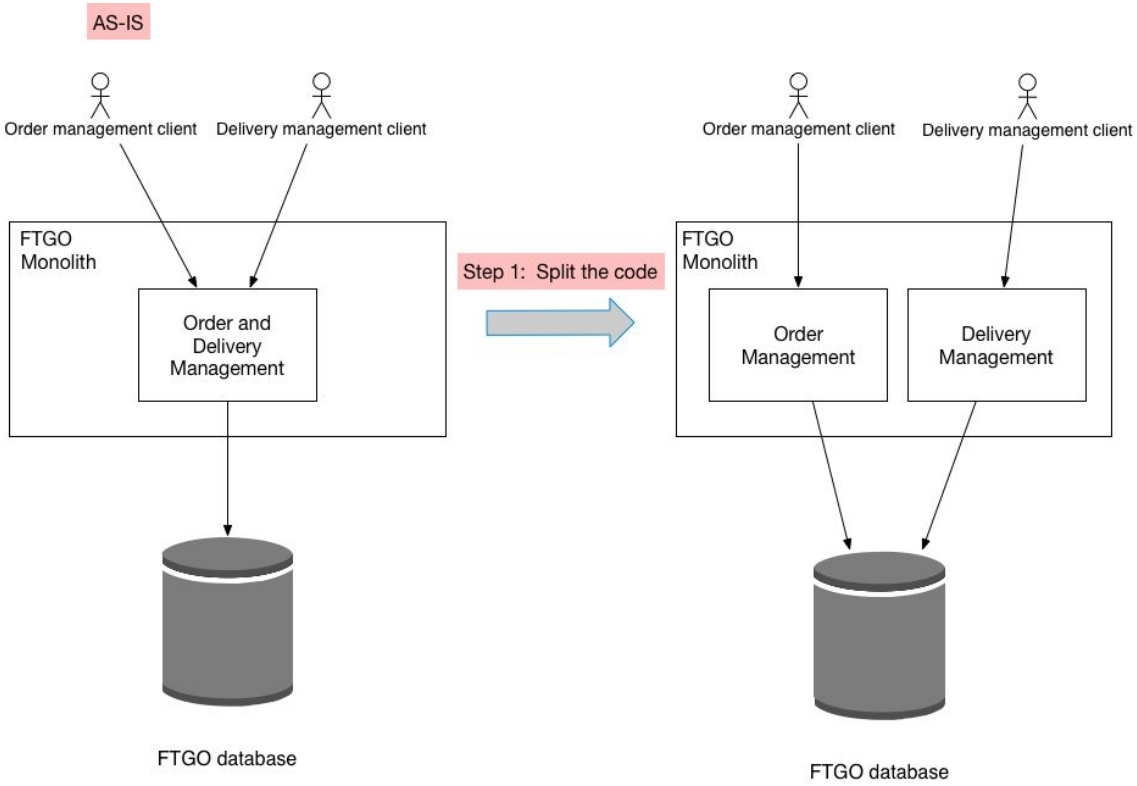
While implementing new functionality as services is extremely useful, the only way of eliminating the monolith is to incrementally extract modules out of the monolith and convert them into services.



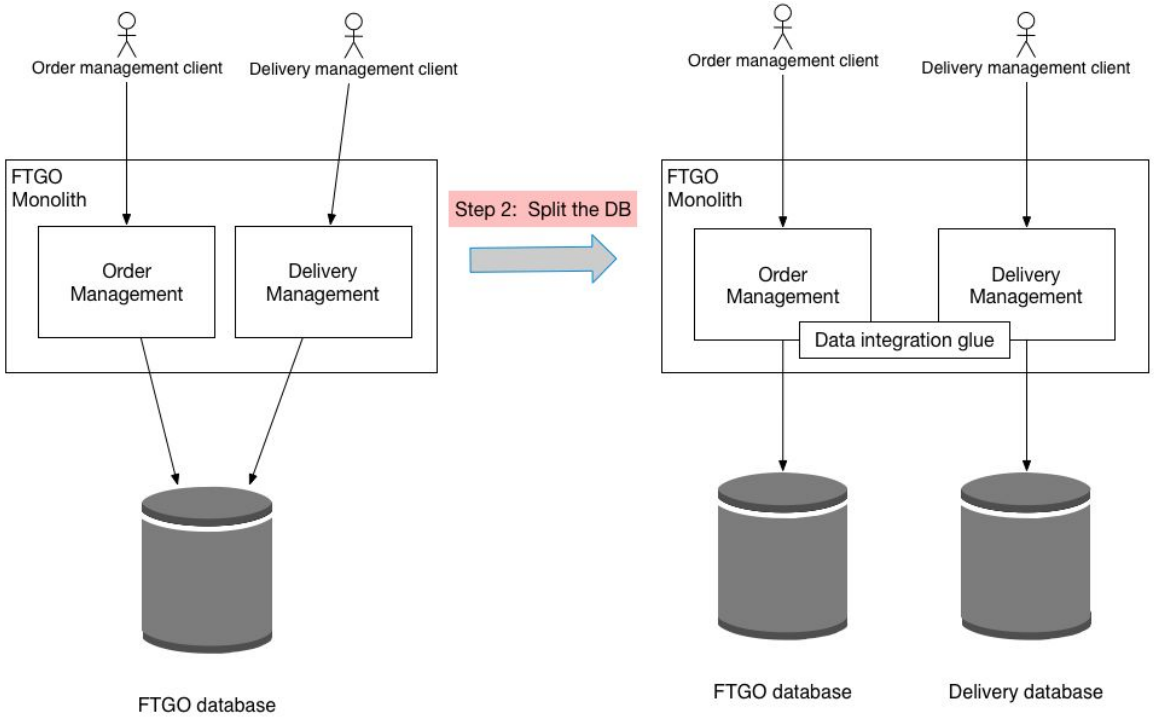
Extracting the Delivery Service consists of the following the steps:

- Split the code and convert delivery management into a separate, loosely coupled module within the monolith
- Split the database and define a separate schema for delivery management.
- Define a standalone **Delivery Service**
- Use the standalone **Delivery Service**
- Remove the old and now unused delivery management functionality from the FTGO monolith

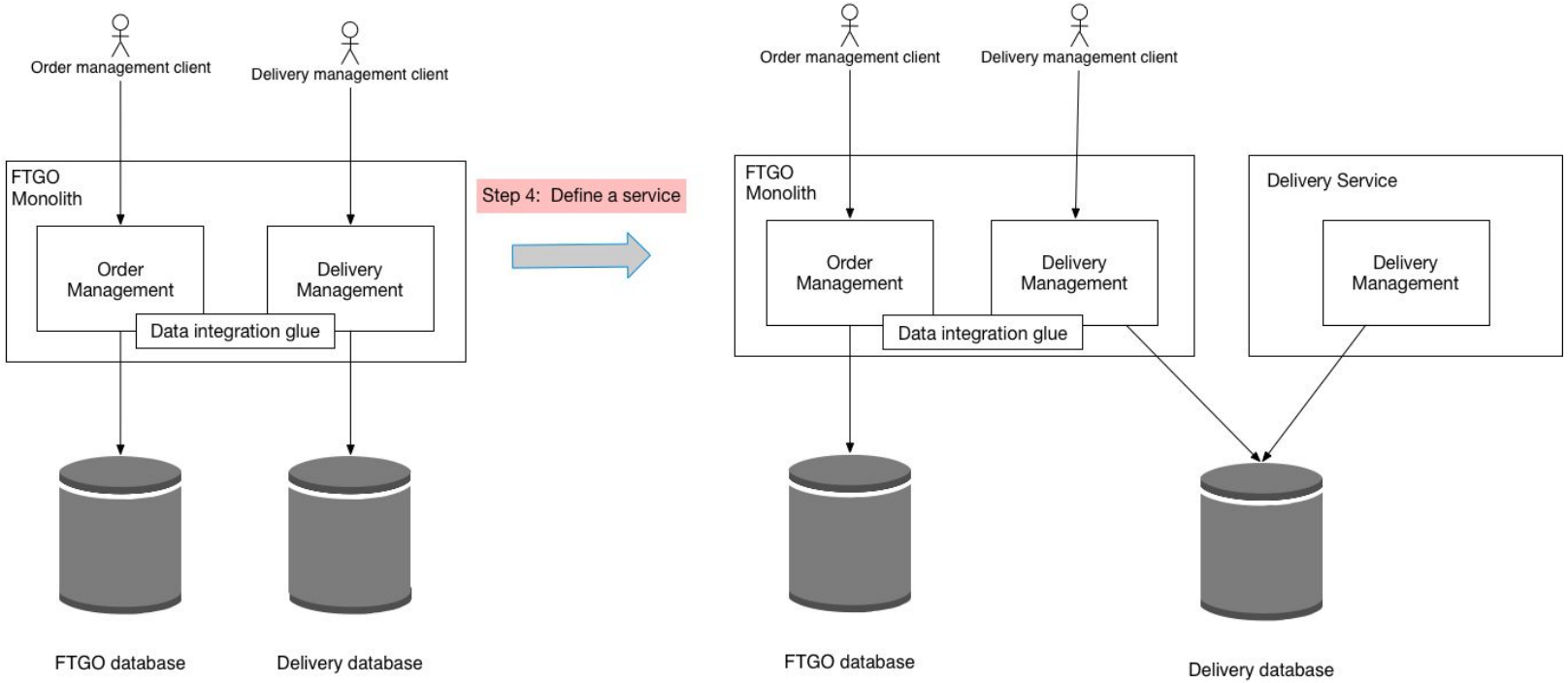
Step 1: Split the code



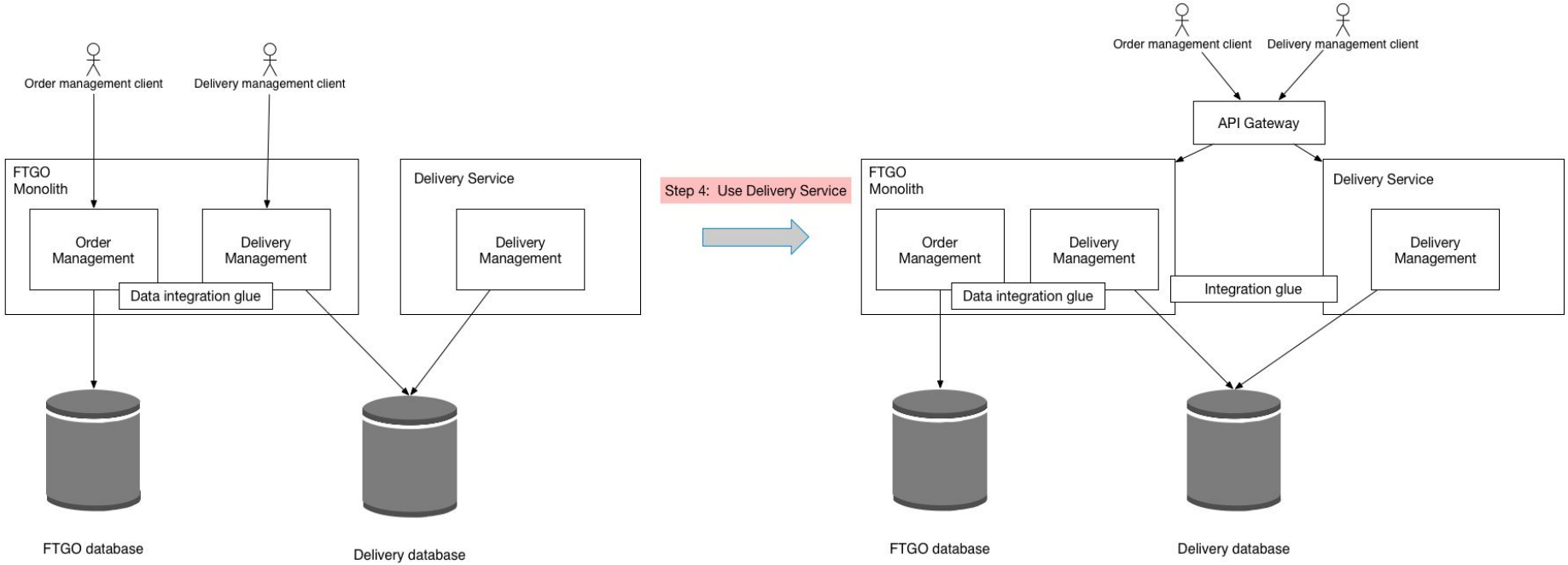
Step 2: Split the database



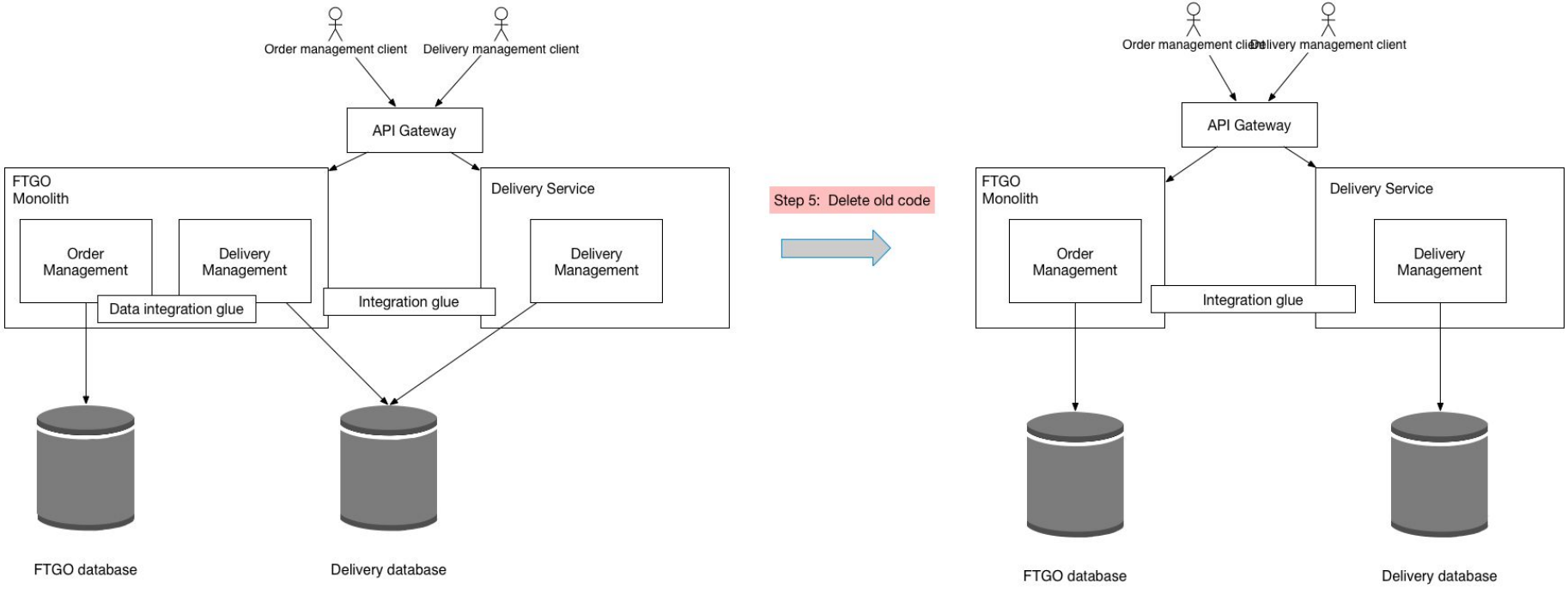
Step 3: Define a standalone Delivery Service



Step 4: Use the standalone Delivery Service



Step 5: Remove the delivery management functionality



Wrapping Up

Migrating existing applications to microservices is intended to enable organizations to realize benefits of microservice architectures, such as resilience, scalability, improved time to market, and easier maintenance, with maximum efficiency and minimal disruption to existing applications and services.

Q&A

NAYA College