Communication and Transaction management

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
Asst. Professor, CSIS Department

innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG583, Scalable Services
Lecture No. 7

Introduction

Let's imagine that you are developing a native mobile client for a shopping application.

The product details page displays a lot of information

- Number of items in the shopping cart
- Order history
- Basic Product Information
- Customer reviews
- Low inventory warning
- Shipping options
- Various suggested items

Direct Communication

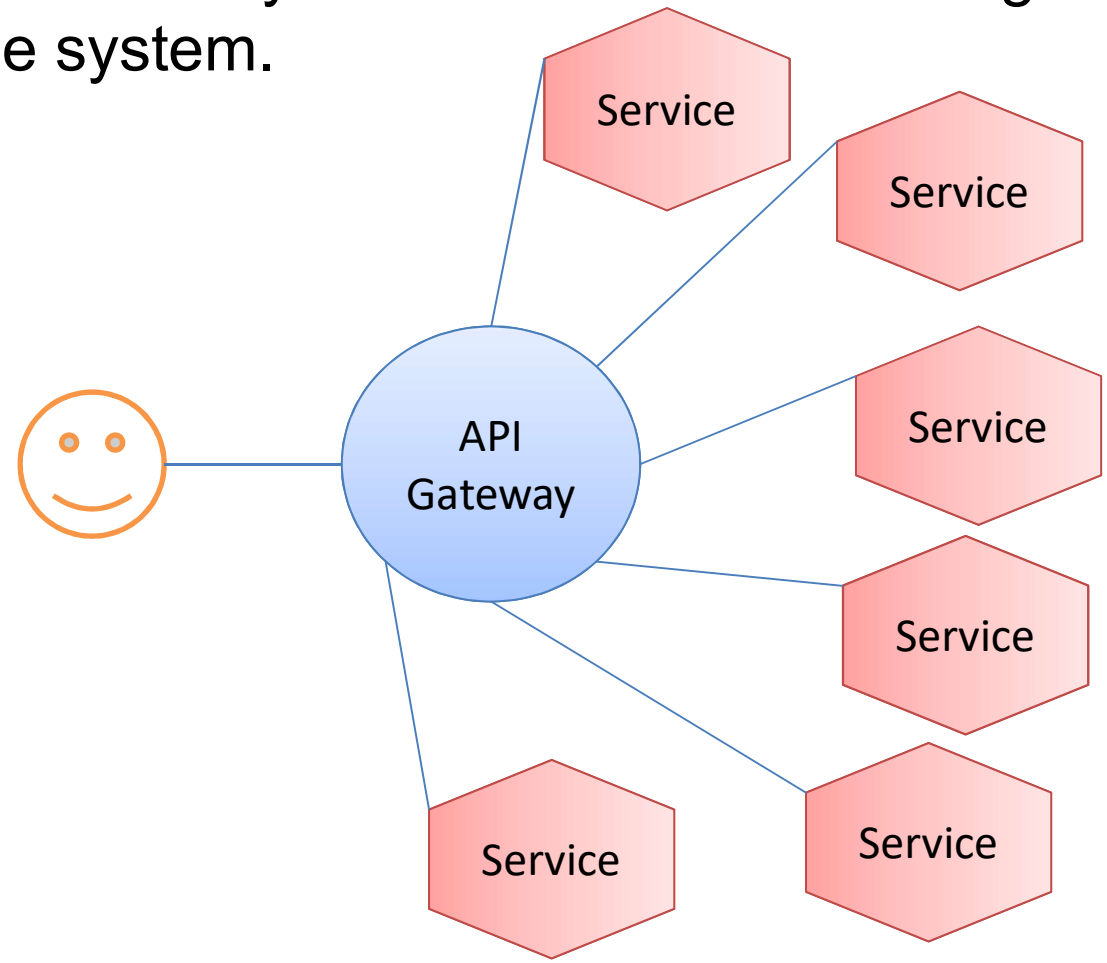
A client could make requests to each of the microservices directly

But there are challenges and limitations with this option

- Inefficient to call directly.
- When the client directly is calling the microservices they might use protocols that are not web-friendly.

Using an API Gateway

An API Gateway is a server that is the single entry point into the system.

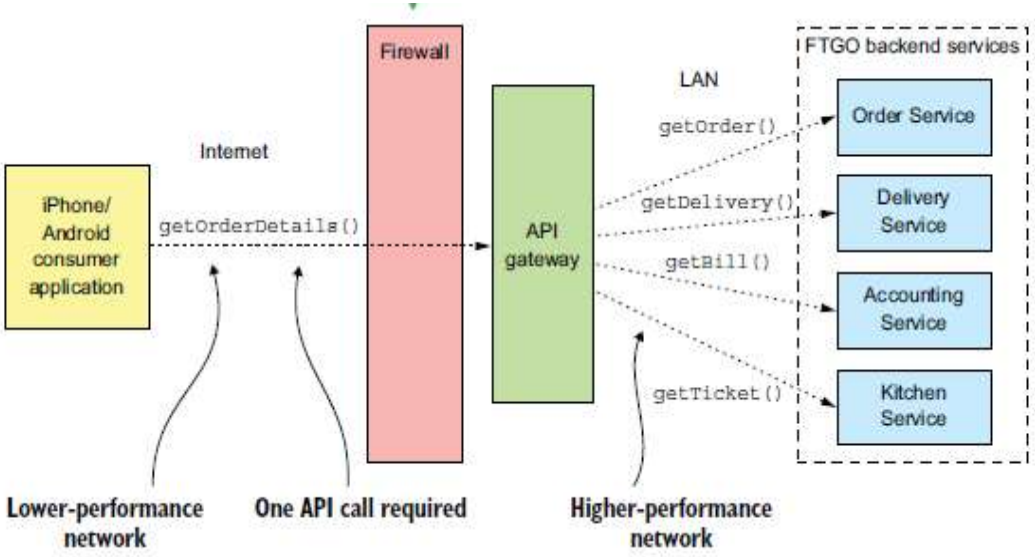


Using an API Gateway: Request Routing

An API gateway implements some API operations by routing requests to the corresponding service.

API Composition

API Gateway might implement some API operations using API composition.



Using an API Gateway

It might provide a RESTful API to external clients

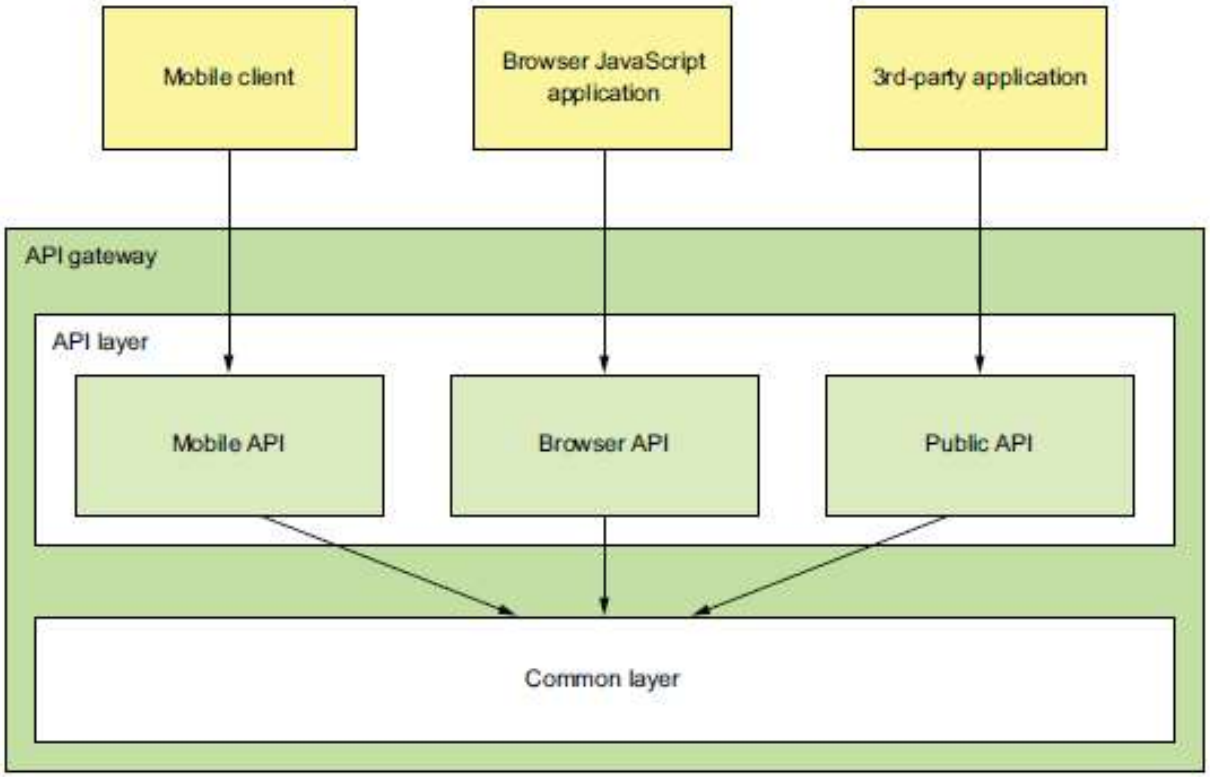
API Gateway provides each client with client-specific API

- Android client and IOS client separate

Implementing edge functions

- Authentication
- Authorization
- Rate limiting
- Caching
- Metrics collection
- Request logging

API Gateway Architecture



Benefits of an API Gateway

- It encapsulates internal structure of the application.
- It also simplifies the client code

Drawbacks of an API Gateway

- Another component that must be developed, deployed, and managed.
- API gateway can become a development bottleneck

What is API design?

API design refers to the process of developing application programming interfaces (APIs) that expose data and application functionality for use by developers and users.

Designing APIs

1. A service's API is a contract between the service and its clients.

2. The challenge is that a service API isn't defined using a simple programming language construct.

3. The nature of the API definition depends on which IPC mechanism you're using.

4. APIs invariably change over time as new features are added, existing features are changed, and old features are removed

How do we design our API program

Teams must ask themselves:

- What technology is used to build the APIs?
- How are the APIs designed?
- How are the APIs maintained?
- How are the APIs promoted inside the organization or marketed to the outside world?
- What resources are available?
- Who should be on the team?

Good API design

1. Simplicity
2. Flexibility
3. Consistency
4. Scalability

External API design issues

- One approach to API design is for clients to invoke the services directly.
- But this approach is rarely used in a microservice architecture because of a lot of drawbacks

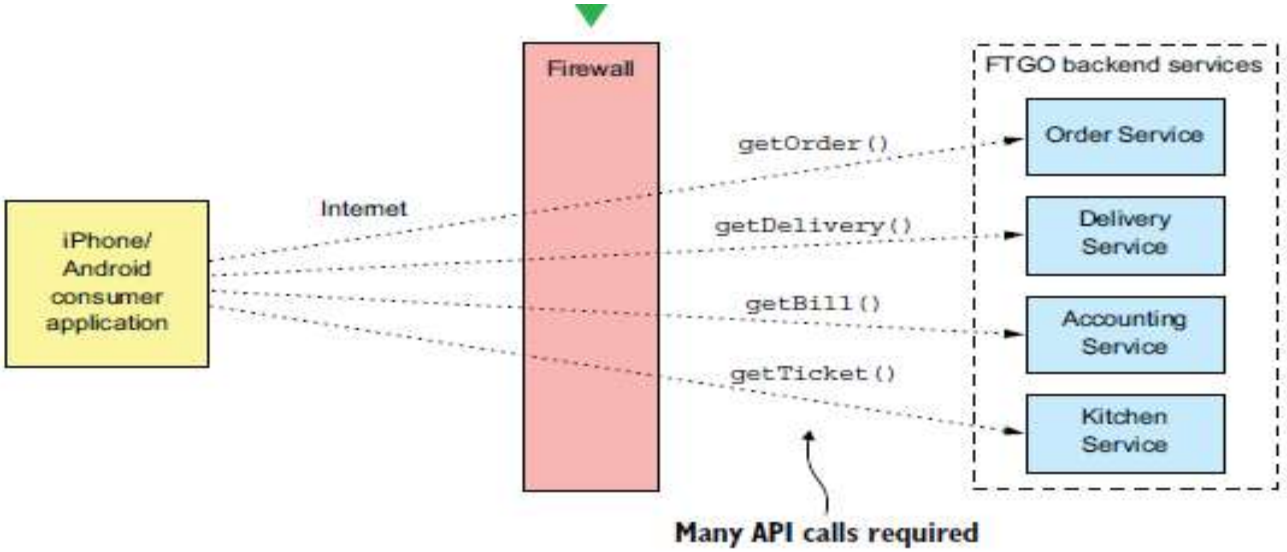


Figure 8.2 A client can retrieve the order details from the monolithic FTGO application with a single request. But the client must make multiple requests to retrieve the same information in a microservice architecture.

What Is The API Contract?

The API contract is the agreement between the API producer and consumer.

An API contract is also a way to keep track of the changes made to an API.

When To Create An API Version

API versioning should occur any time you:

- Change fields or routing after an API is released.
- Change payload structures, such as changing a variable from an integer to float, for instance.
- Remove endpoints to fix design or poor implementation of HTTP.

What is API security?

- API security is the protection of the integrity of APIs, both the ones you own and the ones you use.

Why is API security important?

- Broken, exposed, or hacked APIs are behind major data breaches.

- If your API connects to a third party application, understand how that app is funneling information back to the internet.

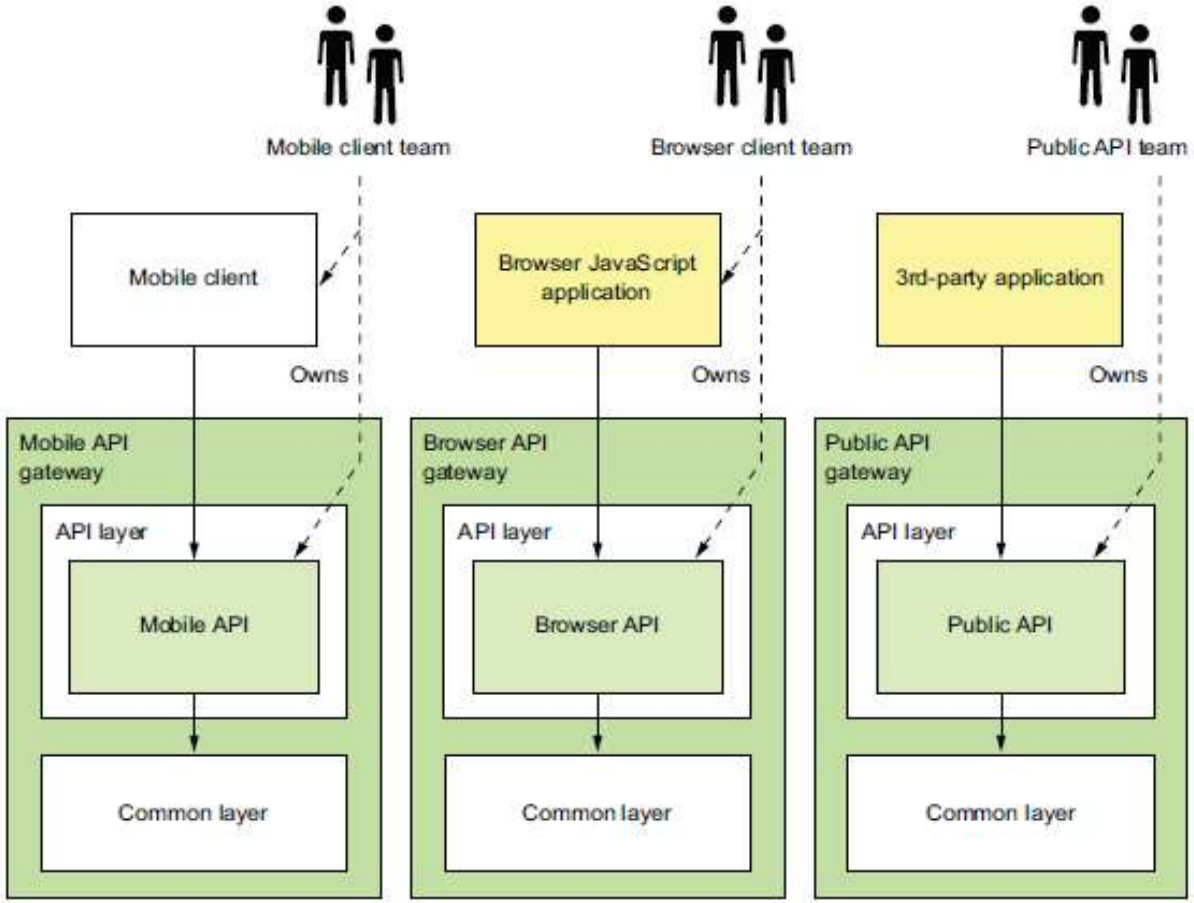
REST API security

- REST APIs use HTTP and support Transport Layer Security (TLS) encryption.
- TLS is a standard that keeps an internet connection private and checks that the data sent between two systems (a server and a server, or a server and a client) is encrypted and unmodified.

Most common API security best practices

- Use tokens
- Use encryption and signatures
- Identify vulnerabilities
- Use quotas and throttling
- Use an API gateway

Backends For Frontends



Benefits of BFF pattern

- The API modules are isolated from one another, which improves reliability.
- It also improves observability, because different API modules are different processes
- Another benefit of the BFF pattern is that each API is independently scalable.
- The BFF pattern also reduces startup time because each API gateway is a smaller, simpler application.

Database related patterns for Microservices

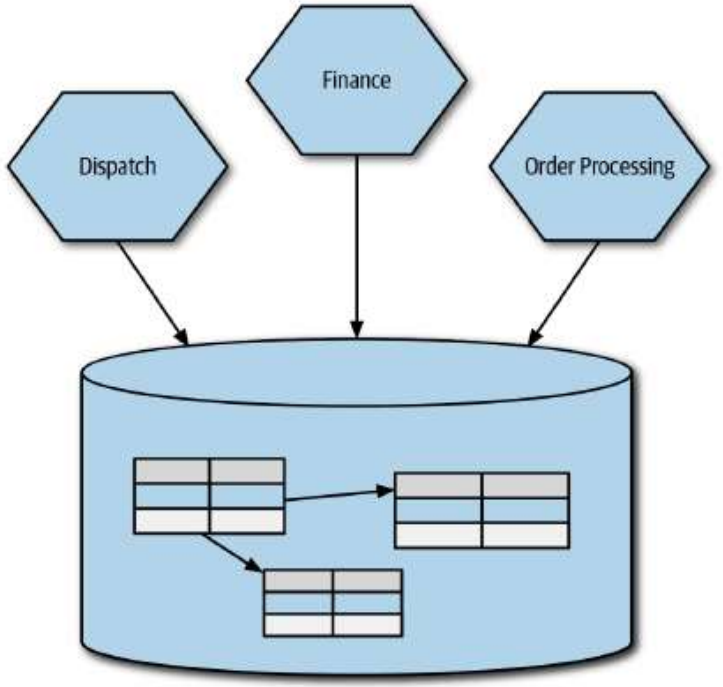
Shared Database pattern

Problems:

- Shared Vs hidden data
- Control on data

When to use:

- Read-only static reference data
- During transition

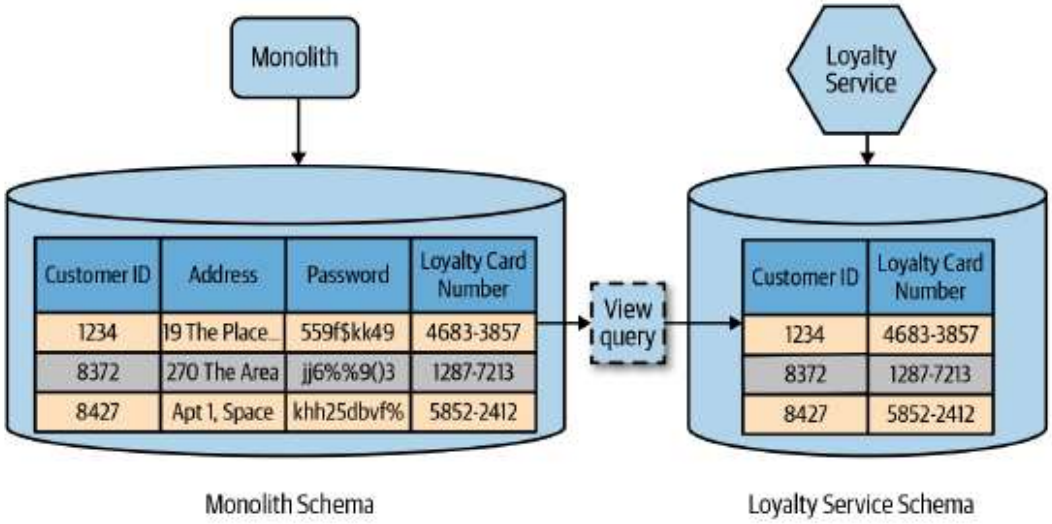


Database View Pattern

- For a single source of data for multiple services, a view can be used to mitigate the concerns regarding coupling

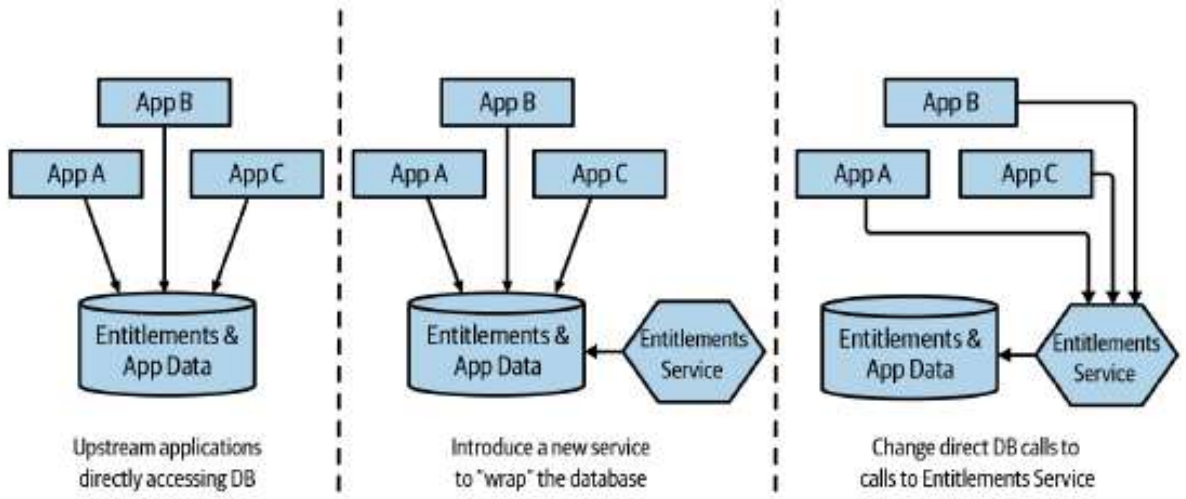
Limitations:

- Views are read-only.



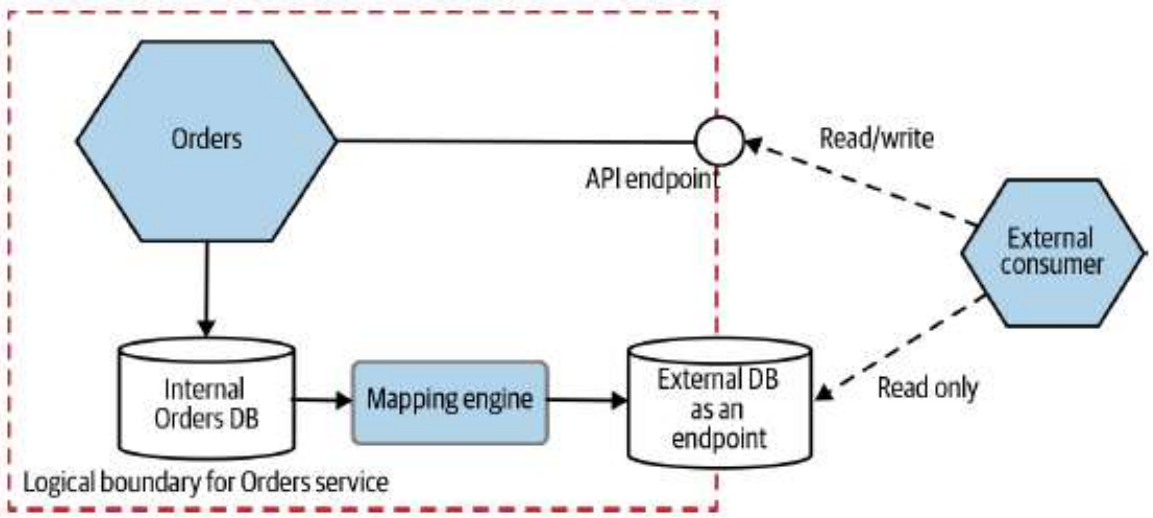
Database Wrapping Service Pattern

- Here we hide the database behind a service that acts as a thin wrapper



Database-as-a-Service Pattern

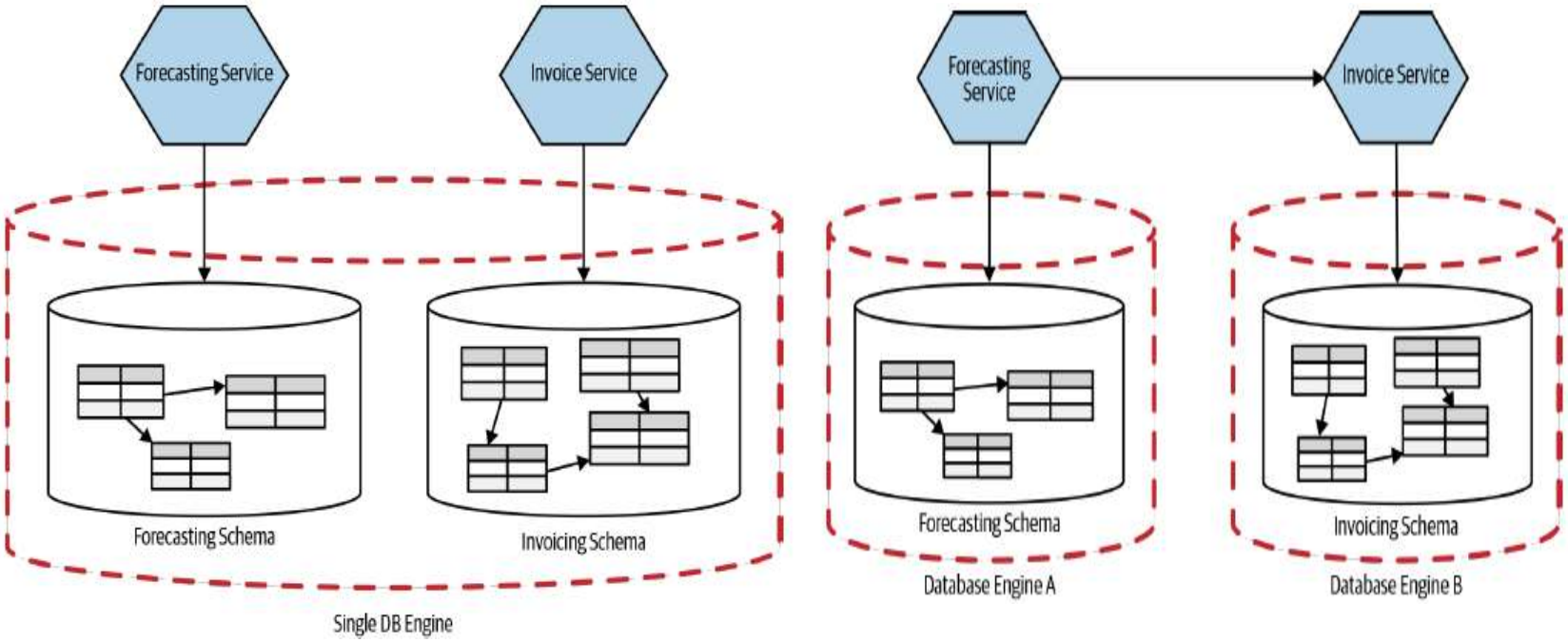
Can be used when clients just need a database to query.



Splitting Apart the Database

innovate achieve lead

Physical Versus Logical Database Separation



Transaction management in a Microservice architecture

Need for distributed transactions in a MSA

- Imagine that you're the FTGO developer responsible for implementing the create- Order() system operation

- this operation must verify that the consumer can place an order, verify the order details, authorize the consumer's credit card, and create an Order in the database

Distributed Transactions

A **distributed transaction** is a transaction that involves multiple networked databases or services, ensuring data consistency across all participating systems. It typically spans multiple nodes in a distributed system, requiring coordination to maintain **ACID (Atomicity, Consistency, Isolation, Durability)** properties.

Key Concepts of Distributed Transactions:

Two-Phase Commit (2PC)

Prepare Phase: The coordinator asks all participants if they can commit the transaction.

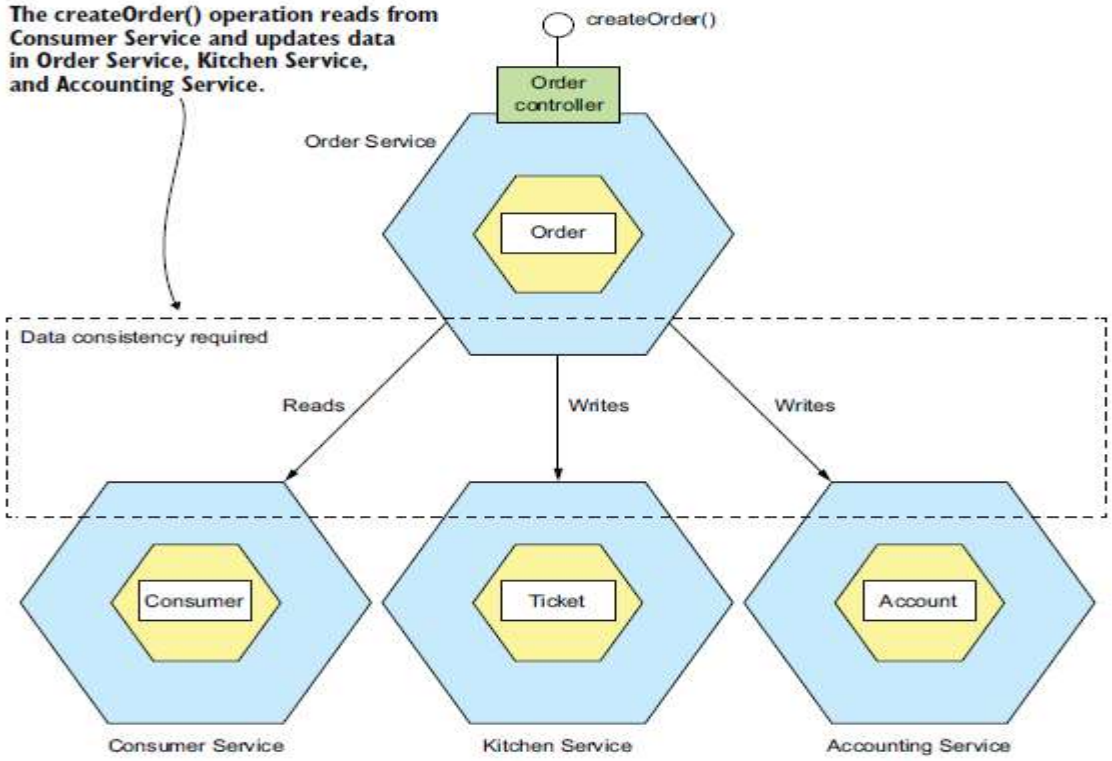
Commit Phase: If all participants agree, the coordinator commits; otherwise, it aborts.

Three-Phase Commit (3PC)

Adds an extra phase to minimize blocking issues in 2PC.

Challenges with distributed transactions

- The traditional approach to maintaining data consistency across multiple services, databases, or message brokers is to use distributed transactions

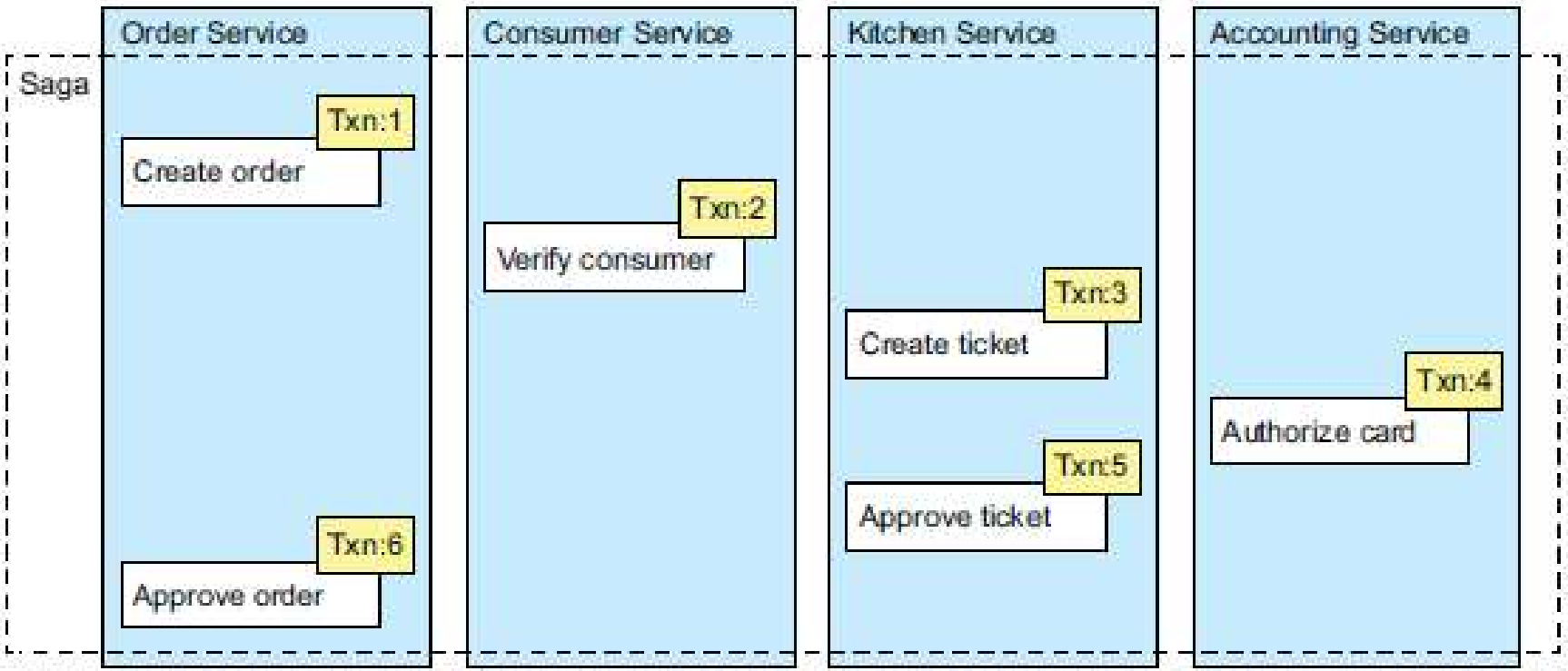


- One problem is that many modern technologies, including NoSQL databases such as MongoDB and Cassandra, don’t support them.
- They are a form of synchronous IPC, which reduces availability. In order for a distributed transaction to commit, all the participating services must be available.
- To solve the more complex problem of maintaining data consistency in a microservice architecture, an application must use a different mechanism that builds on the concept of loosely coupled, asynchronous services.

Solution: Saga pattern

- Sagas are mechanisms to maintain data consistency in a microservice architecture without having to use distributed transactions.

Example

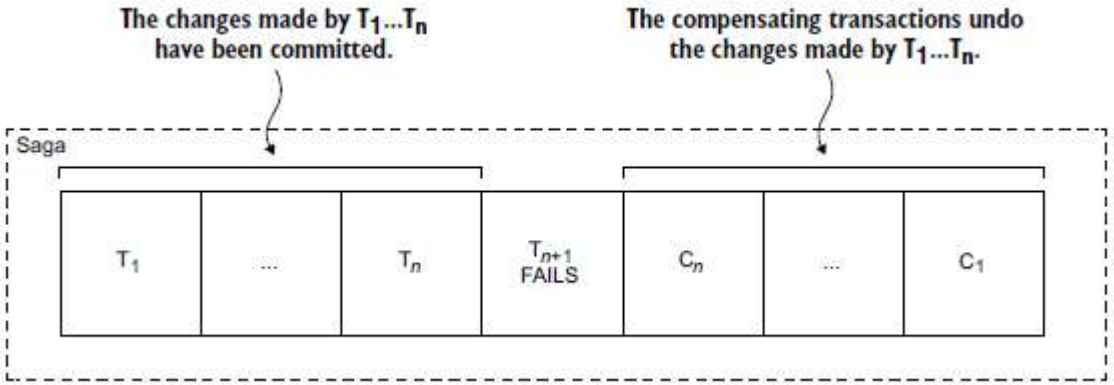

Order Service
Consumer Service
Kitchen Service
Accounting Service
Saga
Txn:1
Create order
Txn:2
Verify consumer
Txn:3
Create ticket
Txn:4
Authorize card
Txn:5
Approve ticket
Txn:6
Approve order

Challenges in Saga

- lack of isolation between sagas
- rolling back changes when an error occurs

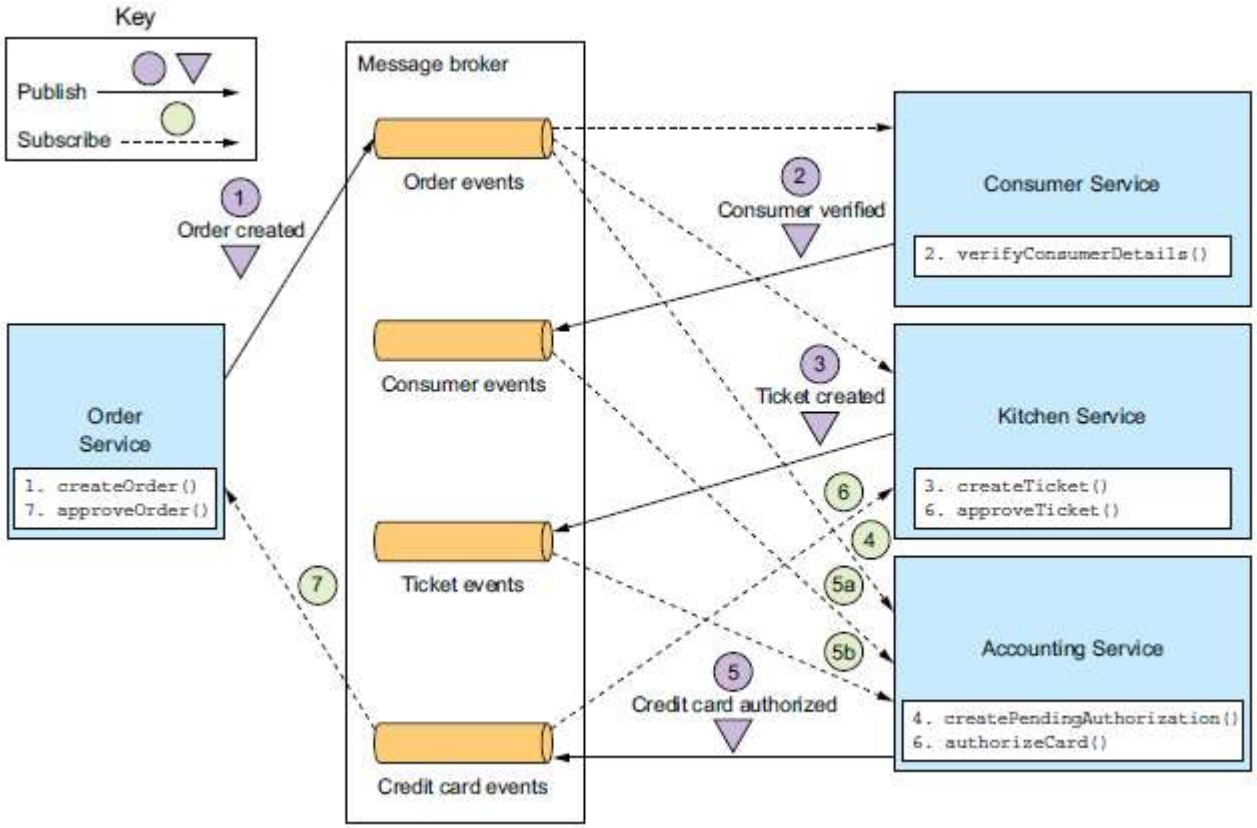
Compensating transactions

- Unfortunately, sagas can't be automatically rolled back, because each step commits its changes to the local database.



Step	Service	Transaction	Compensating transaction
1	`Order Service`	`createOrder()`	`rejectOrder()`
2	`Consumer Service`	`verifyConsumerDetails()`	—
3	`Kitchen Service`	`createTicket()`	`rejectTicket()`
4	`Accounting Service`	`authorizeCreditCard()`	—
5	`Kitchen Service`	`approveTicket()`	—
6	`Order Service`	`approveOrder()`	—

Create Order Saga Using Choreography



Self Study

- https://learn.microsoft.com/en-us/azure/architecture/patterns/saga
- https://www.postman.com/api-platform/api-design/

References

1. Chapter 3 and 4: Microservices Patterns by Chris Richardson
2. Link: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/
3. https://www.redhat.com/en/topics/api/what-is-api-design
4. https://www.redhat.com/en/topics/security/api-security
5. https://marutitech.com/api-gateway-in-microservices-architecture/
6. https://www.getambassador.io/blog/optimizing-microservices-architecture