Hands On - Caching Using Redis

Hands On - Pagination

API Architecture Pattern

RPC

REST

Microservices

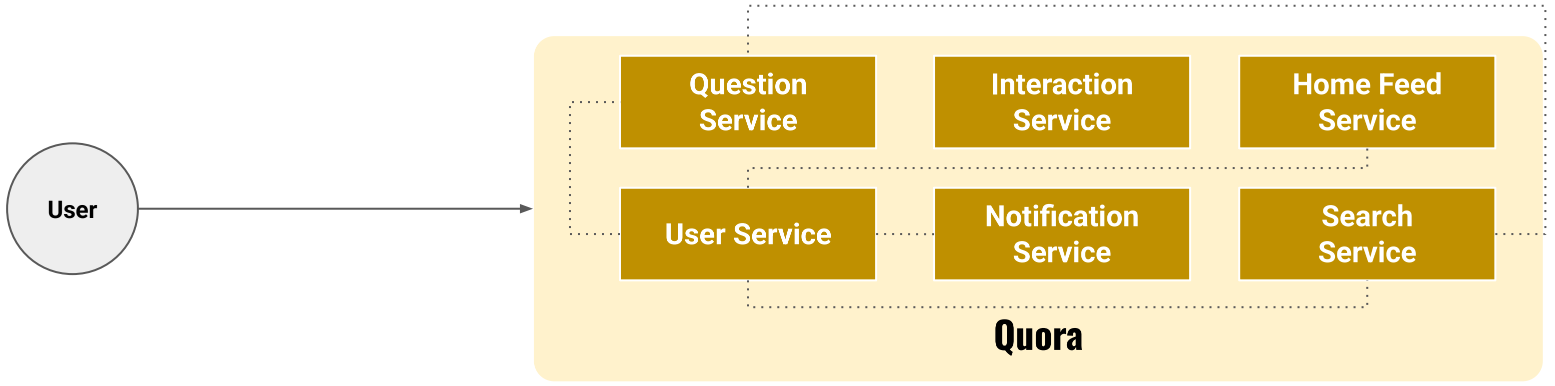
Event Driven

What are microservices?

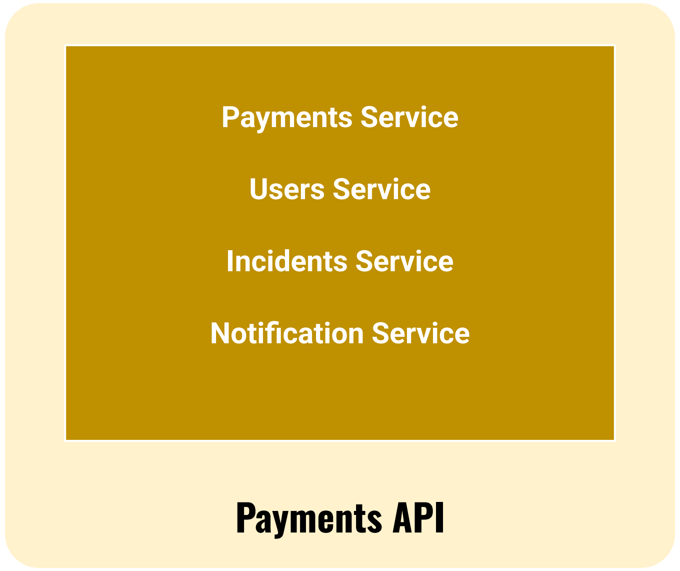
Microservices are small, independently deployed components that deliver one or a small number of digital capabilities/feature

Each service offers one of the many digital feature required, ensuring that each service has a **limited** scope

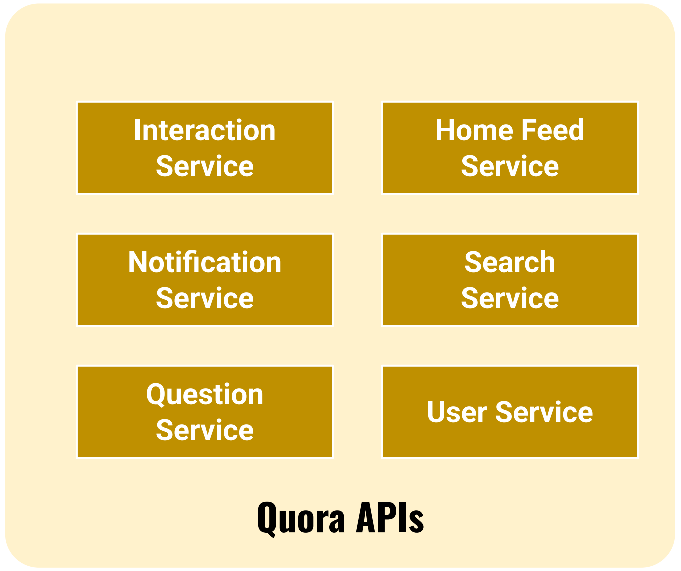
When combined, microservices deliver a highly complex solution using smaller building blocks than the traditional monolith approach



Monolith vs Microservice



Monolith



Microservices

Microservices are not API Implementation Style

We have seen how RPCs help us realize action oriented APIs and REST helps us realize resource oriented APIs. But, Microservices are not an code implementation style

It more of organizing your APIs in modules and making them available as service

In that sense - RPCs can also be deployed as microservices and REST can also be deployed as microservices

Why do we need Microservices?

Monolith APIs or software have some drawbacks. Let’s take a look at them

Scalability

Let’s say our Payments API is becoming popular and traffic has increased. But which functionality do you think will get more traffic? Sign Up/Sing In routes or payments related routes?
So we decide to deploy our API in 3 servers - but since our code in all in one module - we have to deploy the entire in all 3 servers which results in wastage of resources and unnecessary cost.

Instead, if we have microservices - we can just deploy payment services in 3 servers while users services can be in only 1 server

Why do we need Microservices?

Development and Deployment Challenges

Large, complex codebases make overall understanding of the API difficult and thus onboarding new developers harder.
On top of that, it results in different teams working on the same codebase can result in merge conflicts and delays

Every small change and upgrades will need deployment of entire API. This is not only tedious but is extremely risky. **Failure of one module (route) can result in entire API failure.**

For the same reason - experimentation and innovations becomes extremely risky.

Why do we need Microservices?

Technology Lock In

If we have all our routes in single module - then it means all the modules have to be written in **same language**.
Any API will have different areas of work - some will be **transactional** for which we might want to use a highly performant language like **Java**, some might be **analytical** (like search or recommendations) for which languages like **Python** might be a better choice.
This is not very easy to implement in monoliths.

‘Micro’ doesn’t indicate to the size of the code!

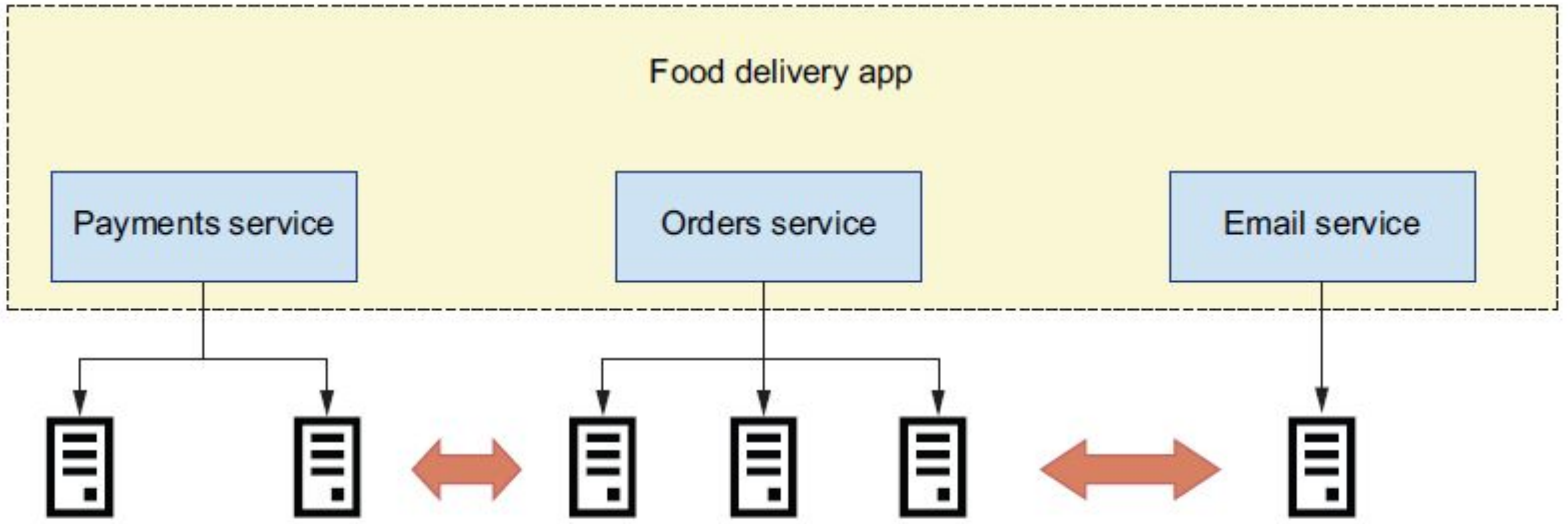
Microservices are small, autonomous services that work together

This definition emphasizes the fact that microservices are applications that run independently of each other yet can collaborate in the performance of their tasks

“Small” doesn’t refer to the size of the code base, but to the idea that microservices are applications with a narrow and well-defined scope, following the Single Responsibility Principle of doing one thing and doing it well

An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API

Microservice Example for a company like Swiggy or Zomato



In microservices architecture, every service implements a specific business subdomain and is deployed as an independent component that runs in its own process

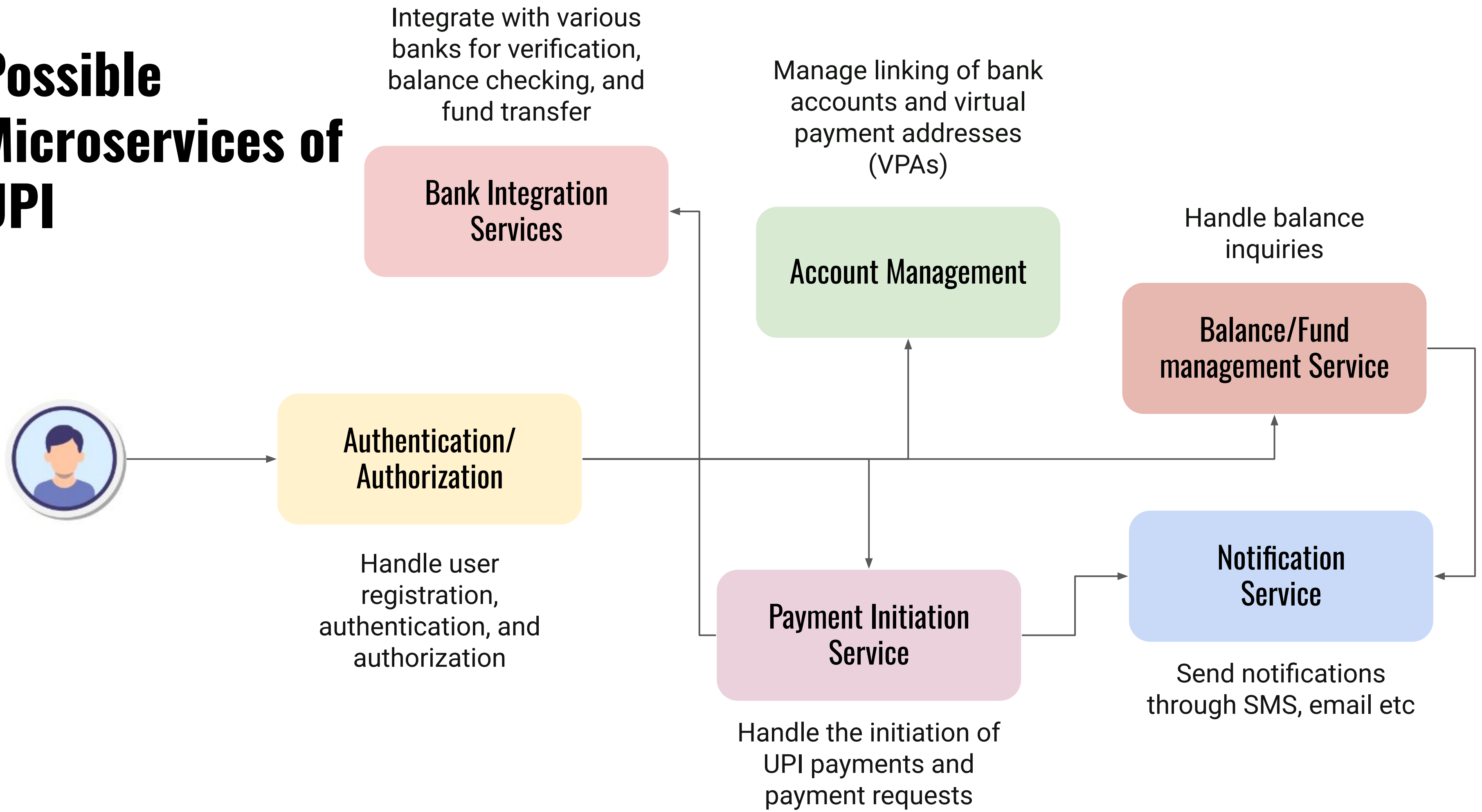
5 mins

<<05:00>>

Create a list of Possible Microservices of UPI

And their possible function or responsibilities

Possible Microservices of UPI


Integrate with various banks for verification, balance checking, and fund transfer
Bank Integration Services
Manage linking of bank accounts and virtual payment addresses (VPAs)
Account Management
Handle balance inquiries
Balance/Fund management Service
Authentication/ Authorization
Handle user registration, authentication, and authorization
Payment Initiation Service
Handle the initiation of UPI payments and payment requests
Notification Service
Send notifications through SMS, email etc

Introduction to Microservice API

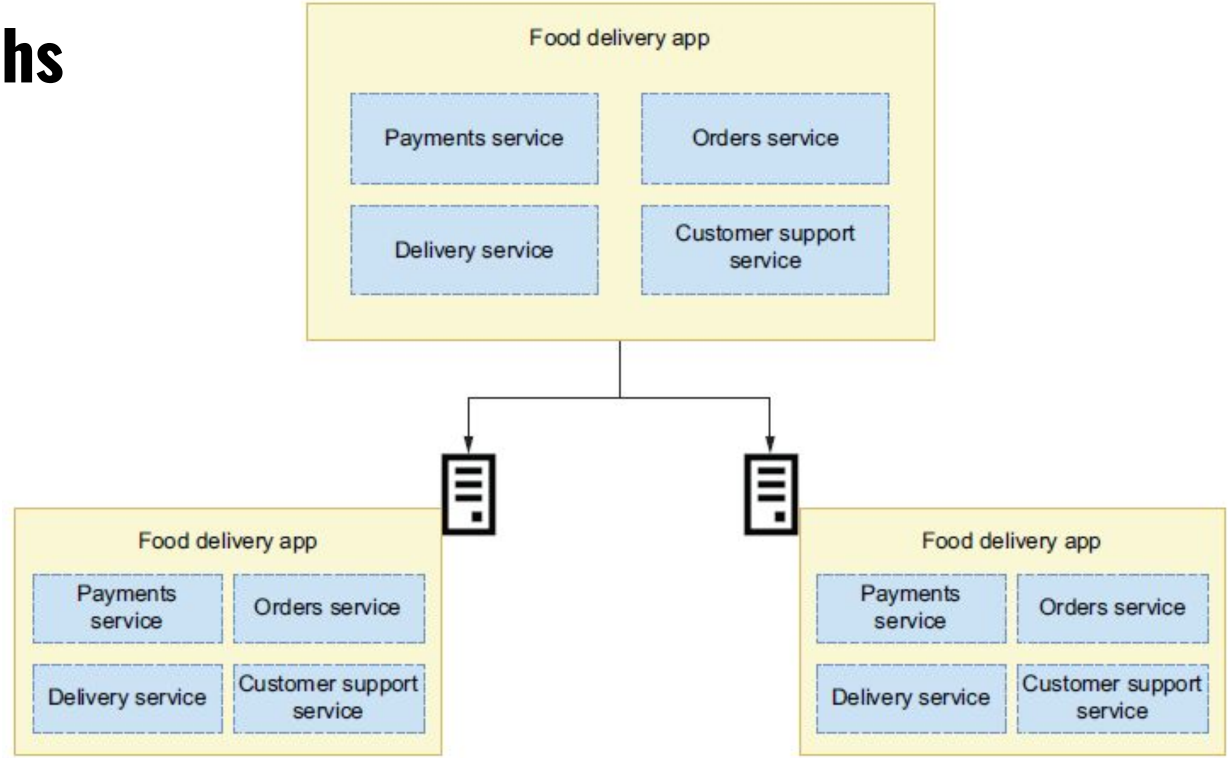
Microservices are an architectural style in which components of a system are designed as independently deployable services

APIs are the interfaces that allow us to interact with those services

In this section, we will see the defining features of microservices architecture and how they compare with monolithic applications

We also discuss most important challenges that we face when designing, implementing, and operating microservices. This discussion is not to deter you from embracing microservices, but so that you can make an informed decision about whether microservices are the right choice of architecture for you

Monoliths



A monolith is a system where all functionality is deployed together as a single build and runs in the same process

Monoliths Might Be The Right Choice For You

Use a monolith when our code base is small and it isn’t expected to grow very large

Having the whole implementation in the same code base makes it easier to access data and logic from different subdomains

It is easy to trace errors through the application: you only need to place a few breakpoints in different parts of your code

Tracing error in microservices is such a difficult work that it has become a whole different domain with companies built around them

Microservices: What does it mean to the client?

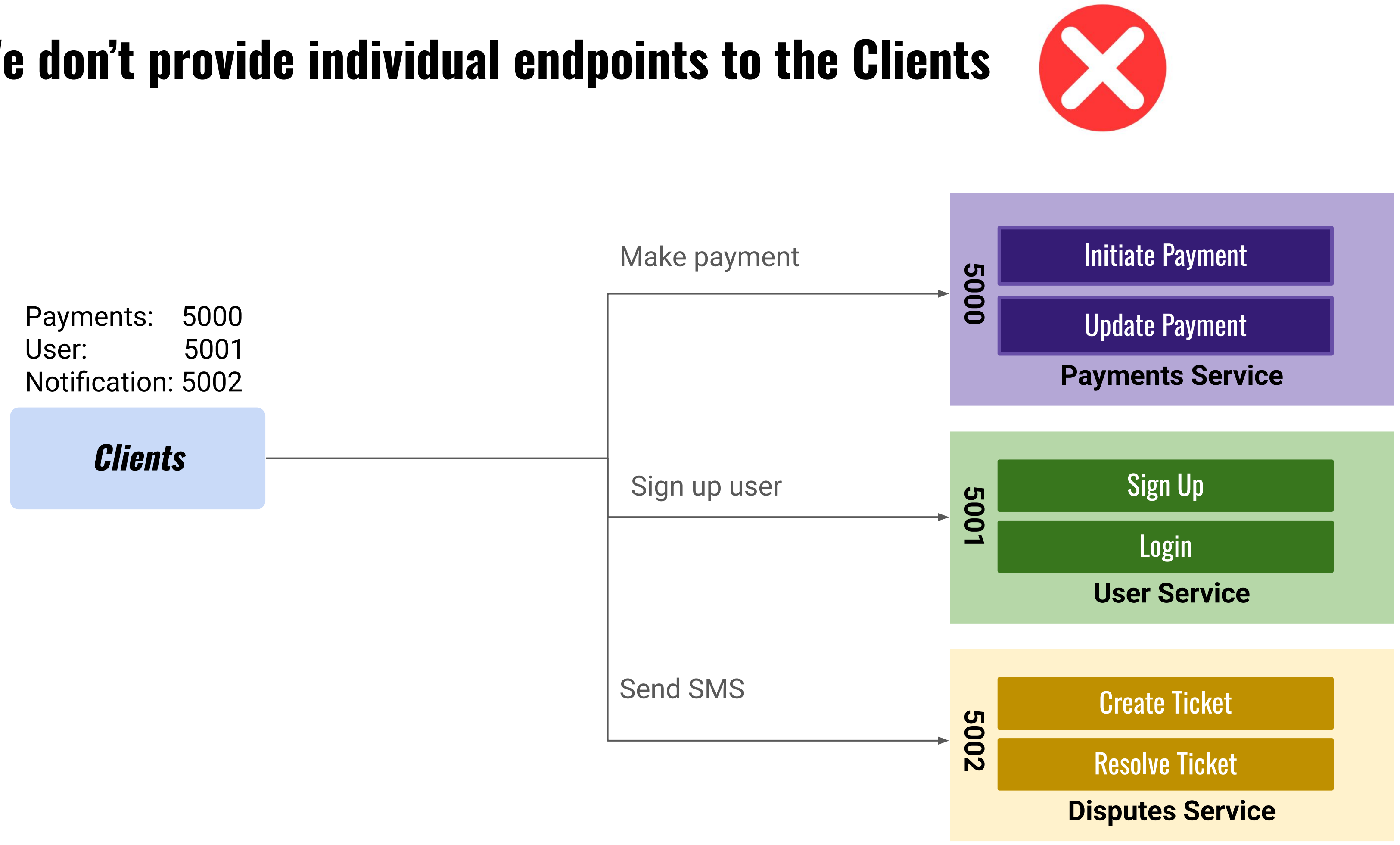
If we have different services running separately like a payment service (create, get, delete, update)
And sign up service (register, un-register, etc): we are going to have multiple endpoints
5000, 5001, 5002

Are we going to give individual endpoints to the client? No!

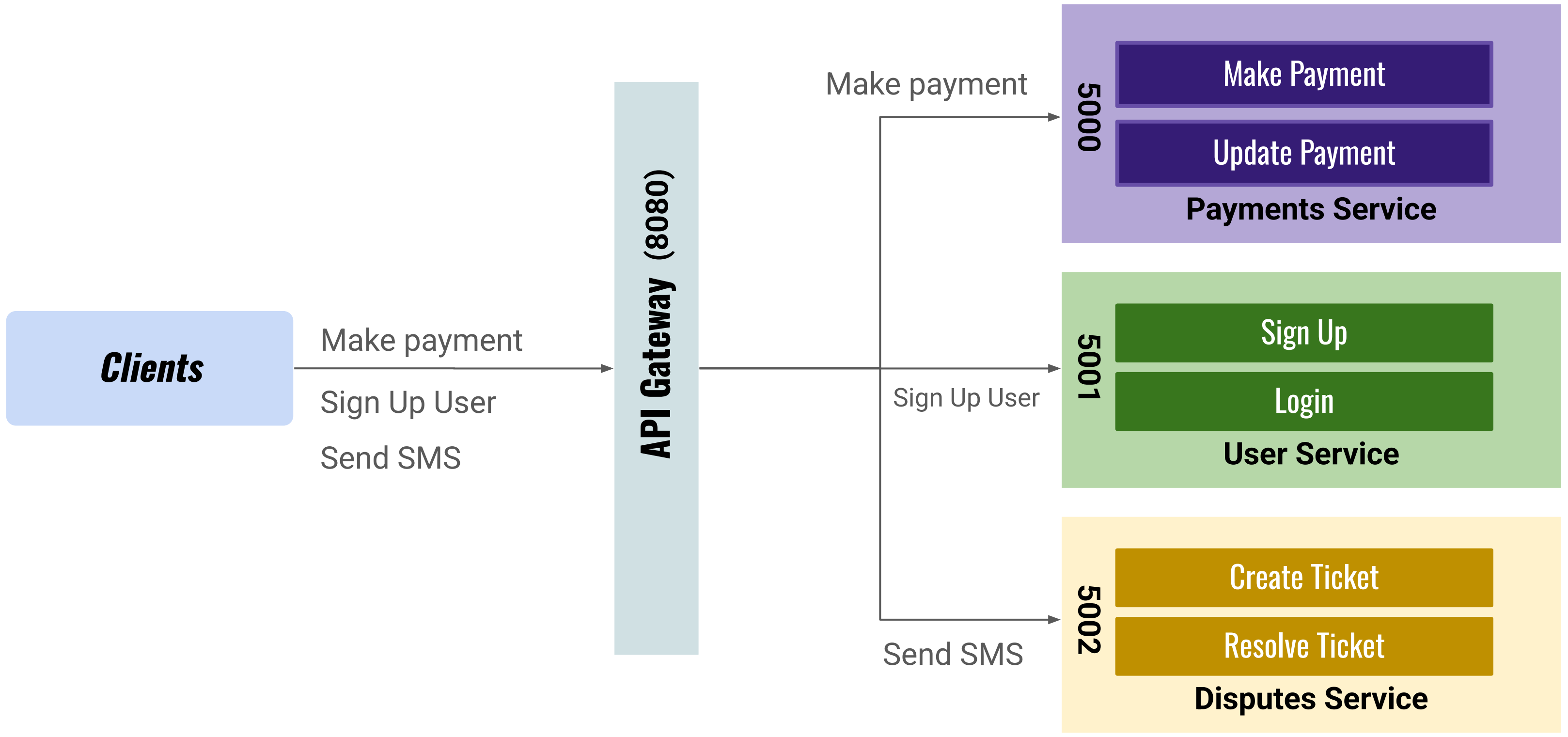
Server details keeps on changing, and we don't want to add the routing complexity on the client side

For those reasons - we will use API gateway that will be one point to entrance to all our APIs

We don’t provide individual endpoints to the Clients



Single Point of Entrance (API Gateway)

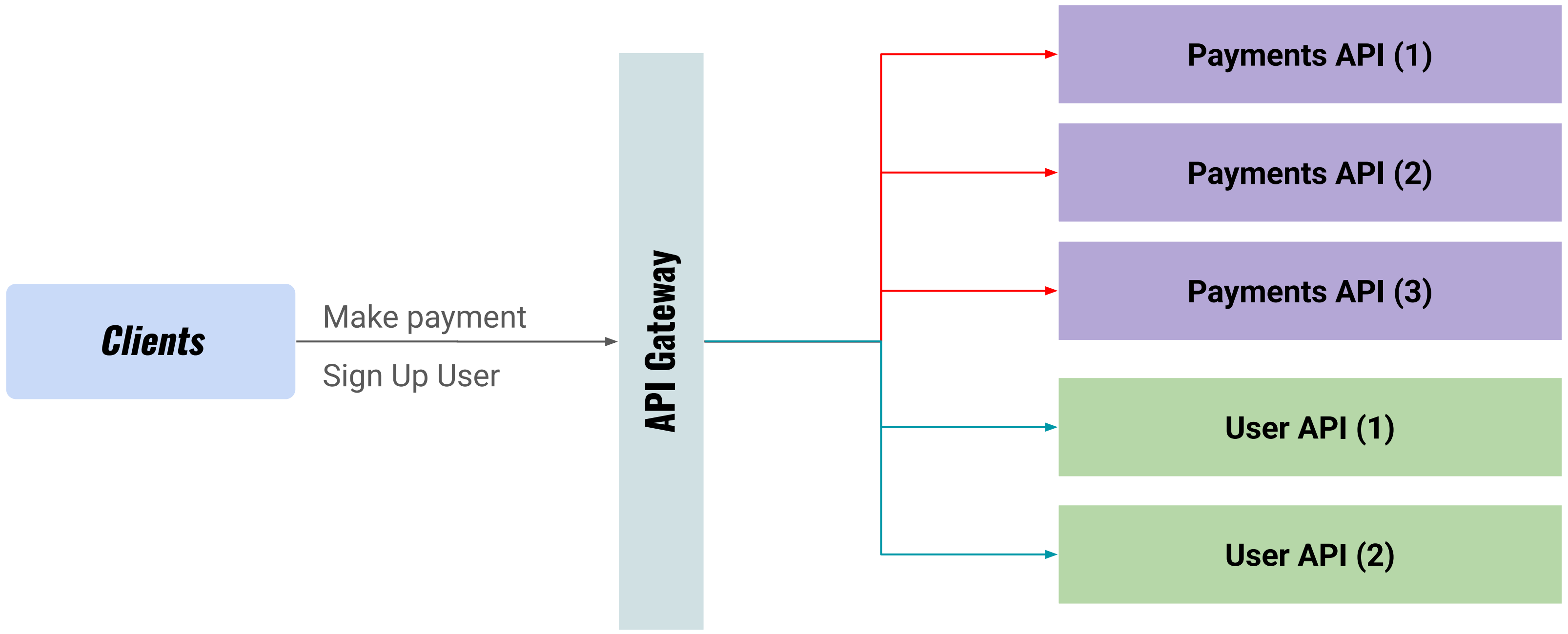


Hands On

- Create a User Microservice (Signup, login, de-register)
- Create a Dispute microservice (Create ticket, update, resolve tiket)
- Distribute traffic among microservices

Link

Load Balancing (Hands On)



Challenges of Microservice Architecture

Microservices Challenges

Microservices bring substantial benefits

However, they also come with significant challenges

- Effective service decomposition
- Microservices integration tests
- Handling service unavailability
- Tracing distributed transactions
- Increased operational complexity and infrastructure overhead

Challenge 1: Effective Service Decomposition

One of the most important challenges when designing microservices is service decomposition

We must break down a platform into loosely coupled yet sufficiently independent components with clearly defined boundaries

You can tell whether you have unreasonable coupling between your services if you find yourself changing one service whenever you change another service

In such situations, either the contract between services is not resilient, or there are enough dependencies between both components to justify merging them

Challenge 2: Microservices Integration Test

In last section we said that microservices are usually easier to test, and that their individual test suites generally run faster

Microservices integration tests, however, can be significantly more difficult to run, especially in cases where a single transaction involves collaboration among several microservices

When your whole application runs within the same process, it is fairly easy to test the integration between different components

The difficulty gets even worse when different microservices are built by different team and run on different machines/networks, etc

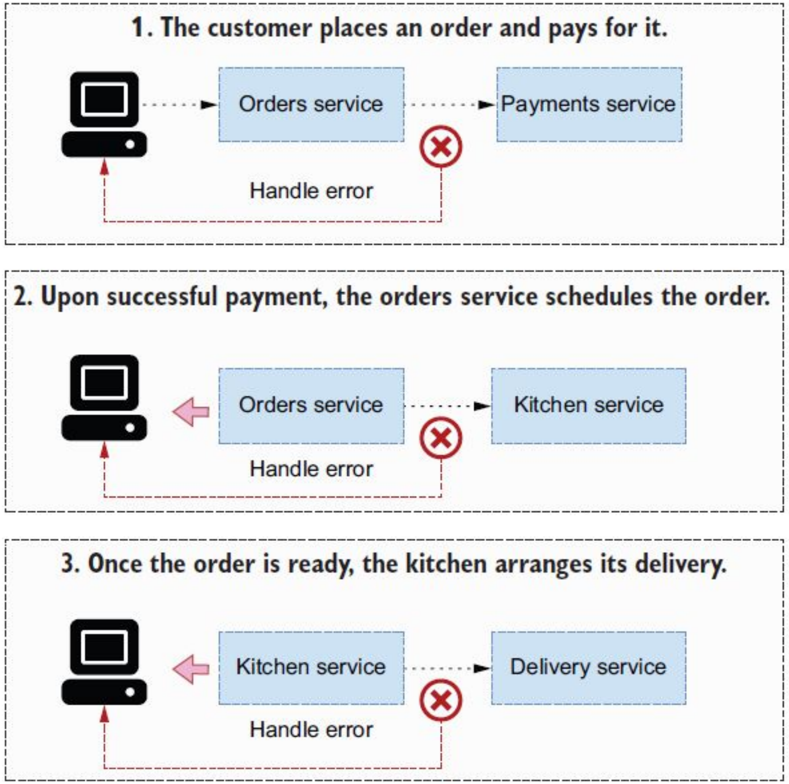
Challenge 3: Handling Service Unavailability

We have to make sure that our applications are resilient in the face of service unavailability, connections and request timeouts, erroring requests, etc

For example, when we place an order through a food delivery application such as Swiggy a chain of requests between services unfolds to process and deliver the order, and any of those requests can fail at any point

Let’s take a look at the process or steps that unfurls when we order food

Challenge 3: Handling Service Unavailability

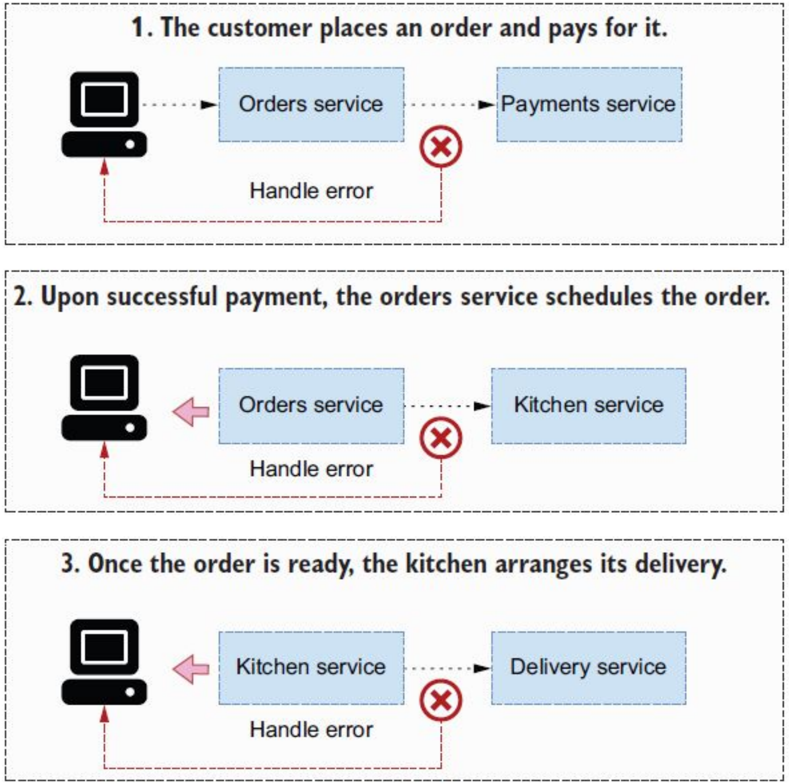

1. The customer places an order and pays for it.
Orders service
Payments service
Handle error
2. Upon successful payment, the orders service schedules the order.
Orders service
Kitchen service
Handle error
3. Once the order is ready, the kitchen arranges its delivery.
Kitchen service
Delivery service
Handle error

1

A customer places an order and pays for it

The order is placed using the orders service, and to process the payment, the orders service work together with the payments service

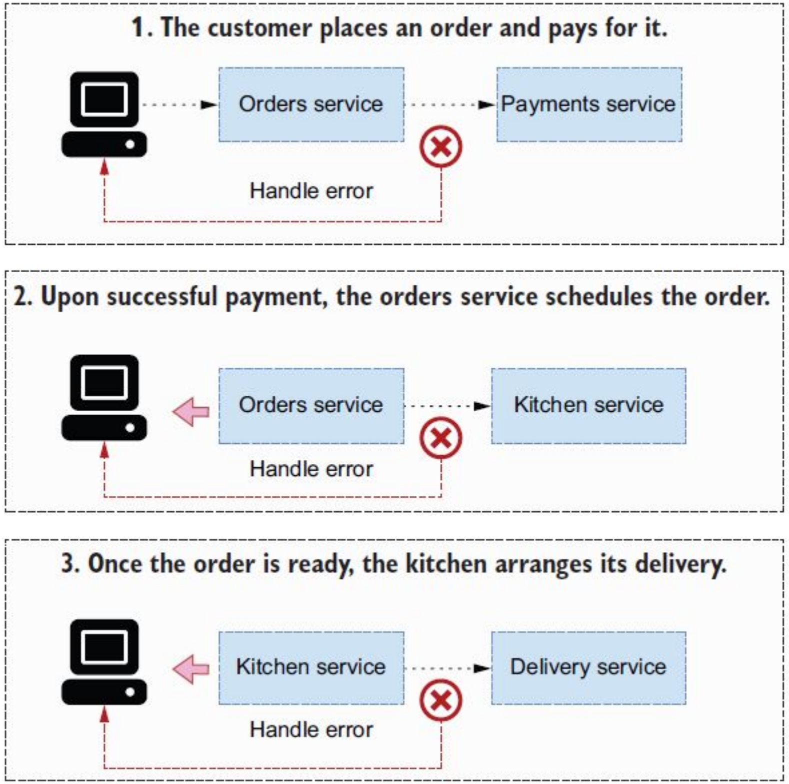
Challenge 3: Handling Service Unavailability


1. The customer places an order and pays for it.
Orders service
Payments service
Handle error
2. Upon successful payment, the orders service schedules the order.
Orders service
Kitchen service
Handle error
3. Once the order is ready, the kitchen arranges its delivery.
Kitchen service
Delivery service
Handle error

2

If payment is successful, the orders service makes a request to the kitchen service to schedule the order for production

Challenge 3: Handling Service Unavailability

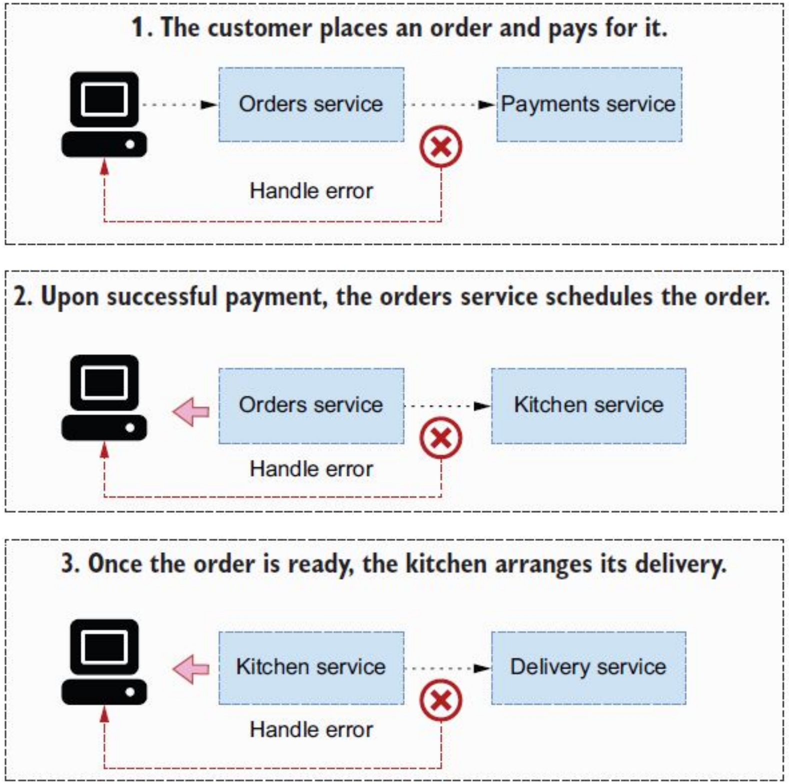


1. The customer places an order and pays for it.
Orders service
Payments service
Handle error
2. Upon successful payment, the orders service schedules the order.
Orders service
Kitchen service
Handle error
3. Once the order is ready, the kitchen arranges its delivery.
Kitchen service
Delivery service
Handle error

3

Once the order has been produced, the kitchen service makes a request to the delivery service to schedule the delivery

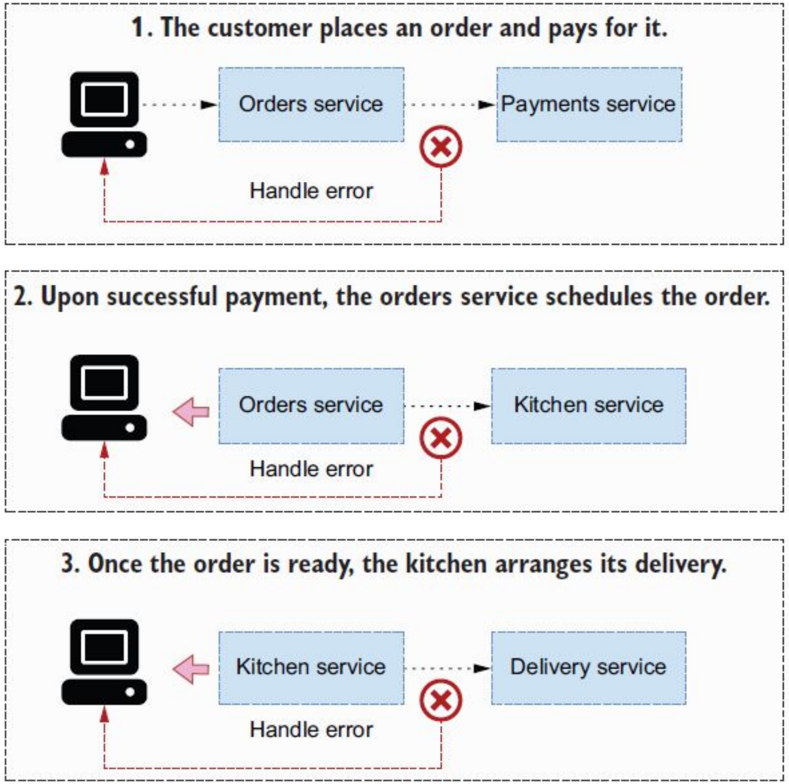
Challenge 3: Handling Service Unavailability


1. The customer places an order and pays for it.
Orders service
Payments service
Handle error
2. Upon successful payment, the orders service schedules the order.
Orders service
Kitchen service
Handle error
3. Once the order is ready, the kitchen arranges its delivery.
Kitchen service
Delivery service
Handle error

In this complex chain of requests, if one of the services involved fails to respond as expected, it can trigger a cascading error through the platform that leaves the order unprocessed or in an inconsistent state

For this reason, it is important to design microservices so that they can deal reliably with failing endpoints. Our end-to-end tests should consider these scenarios and test the behavior of our services in those situations

Challenge 4: Tracing Distributed Transactions


1. The customer places an order and pays for it.
Orders service
Payments service
Handle error
2. Upon successful payment, the orders service schedules the order.
Orders service
Kitchen service
Handle error
3. Once the order is ready, the kitchen arranges its delivery.
Kitchen service
Delivery service
Handle error

Collaborating services must sometimes handle distributed transactions

Distributed transactions are those that require the collaboration of two or more services

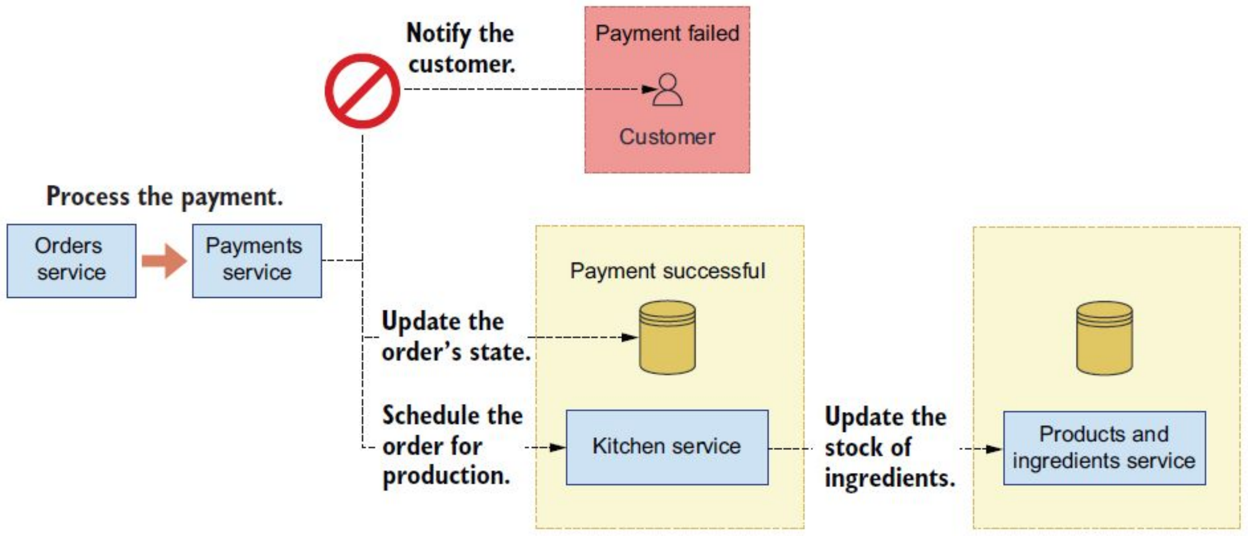
For example, in a food delivery application, we want to keep track of the existing stock of ingredients so that our catalogue can accurately reflect product availability

When a user places an order, we want to update the stock of ingredients to reflect the new availability

Specifically, we want to update the stock of ingredients once the payment has been successfully processed

Challenge 4: Tracing Distributed Transactions

Successful processing of an order involves the following actions



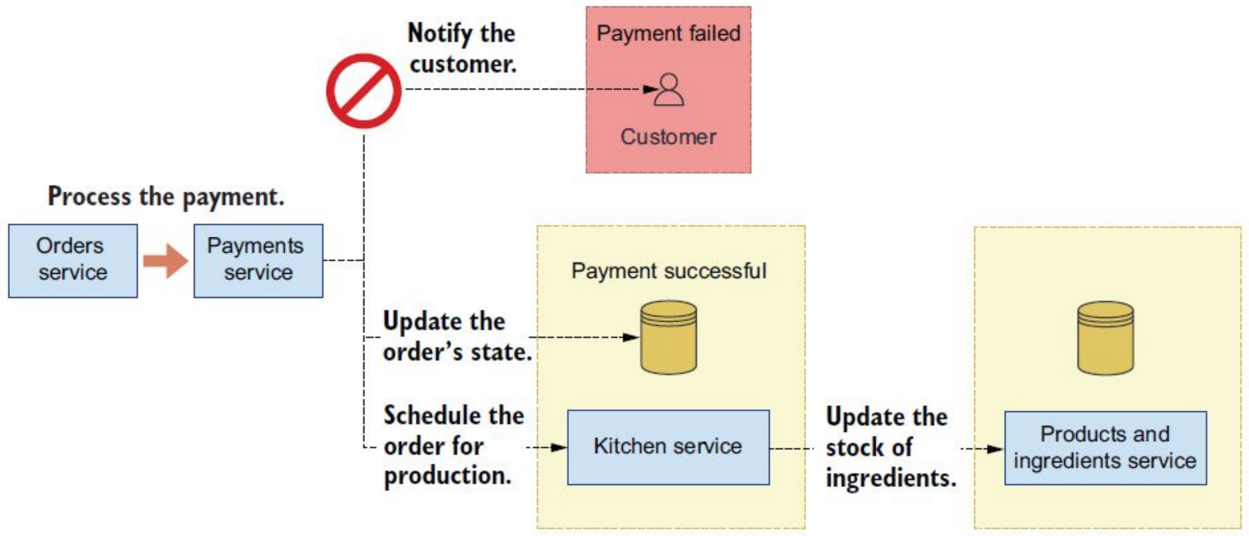
Process the payment

If payment is successful, update the order's status to indicate that it's in progress

Interface with the kitchen service to schedule the order for production

Challenge 4: Tracing Distributed Transactions

Successful processing of an order involves the following actions

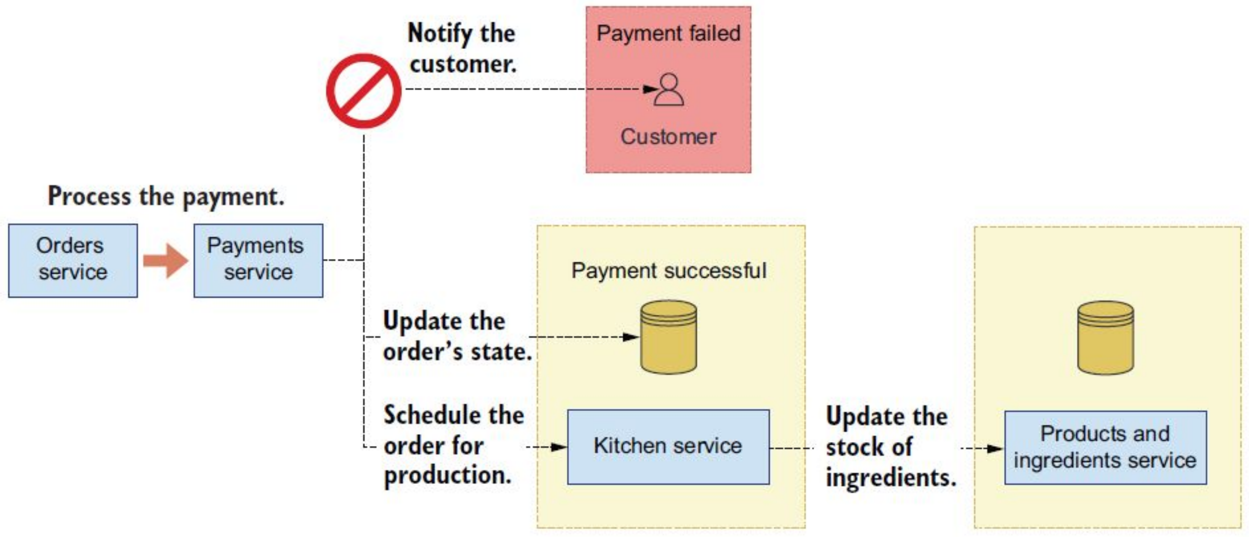


Interface with the kitchen service to schedule the order for production

Update the stock of ingredients to reflect their current availability

Challenge 4: Tracing Distributed Transactions

All of these operations are related, and they must be orchestrated so that they either all succeed or fail together



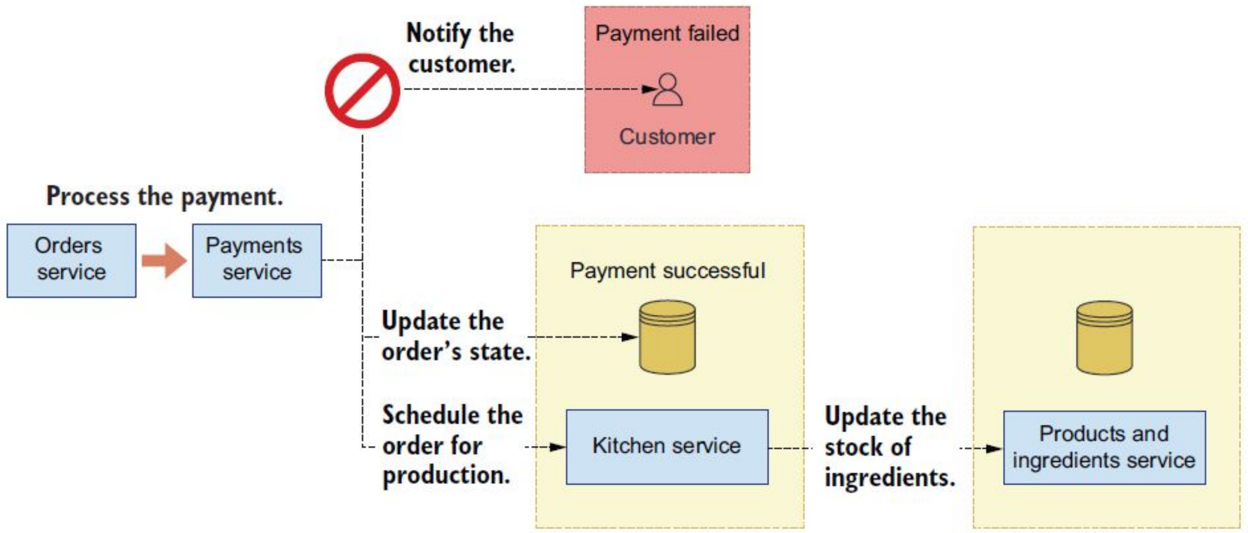
We can't have an order successfully paid without correctly updating its status

And we shouldn't schedule its production if payment fails

We may want to update the availability of the ingredients at the time of making the order, and if payment fails later on, we want to make sure we rollback the update

Challenge 4: Tracing Distributed Transactions

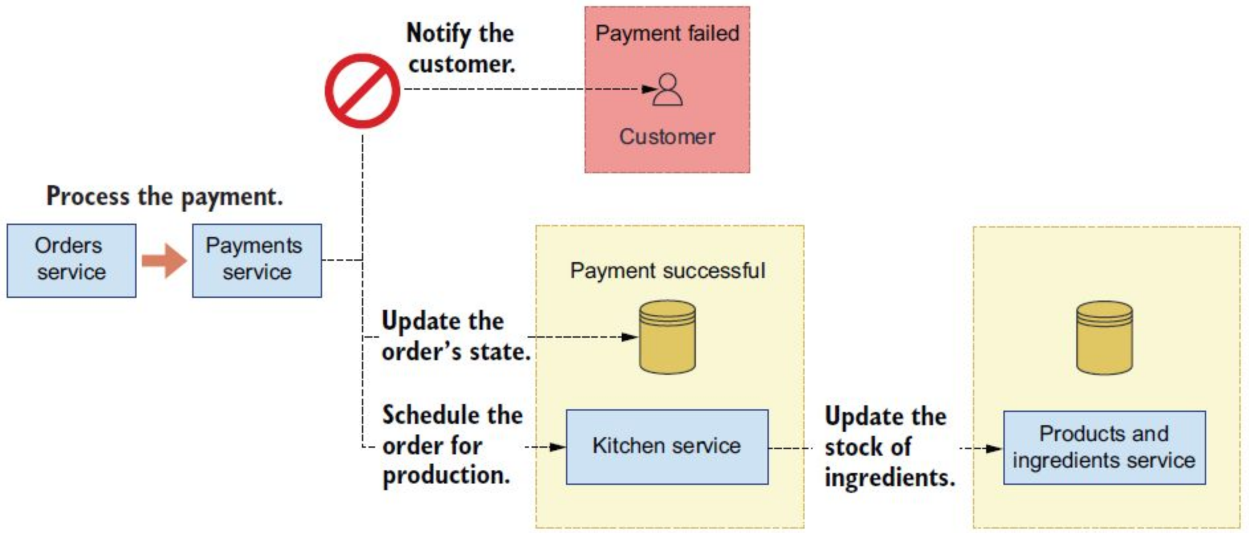
If all these actions happen within the same process, managing the flow is straightforward



But with microservices we must manage the outcomes of various processes

When using microservices, the challenge is ensuring that we have a robust communication process among services so that we know exactly what kind of error happens when it does, and we take appropriate measures in response to it

Challenge 4: Tracing Distributed Transactions



In the case of services that work collaboratively to serve certain requests, you also must be able to trace the cycle of the request as it goes across the different services to be able to spot errors during the transaction

To gain visibility of distributed transactions, you'll need to set up distributed logging and tracing for your microservices

Challenge 5: Increased Operational Complexity and Infra Overhead

Another important challenge that comes with microservices is the increased operational complexity and operational overhead they add to your platform

When the whole backend of your website runs within a single application build, you only need to deploy and monitor one process

When you have a dozen microservices, every service must be configured, deployed, and managed

And this includes not only the provisioning of servers to deploy the services, but also log aggregation streams, monitoring, systems, alerts, self-recovery mechanisms, and so on

Challenge 5: Increased Operational Complexity and Infra Overhead

When Amazon first started their journey toward a microservices architecture, they discovered that development teams would spend about 70% of their time managing infrastructure

This is a very real risk that you face if you do not adopt best practices for infrastructure automation from the beginning

And even if you do, you are likely to spend a significant amount of time developing custom tooling to manage your services effectively and efficiently

Quiz

API Architecture Pattern

RPC

REST

Microservices

Event Driven

Challenges in the communication of Microservices

Due to the benefits of microservices - we implemented our API as microservices

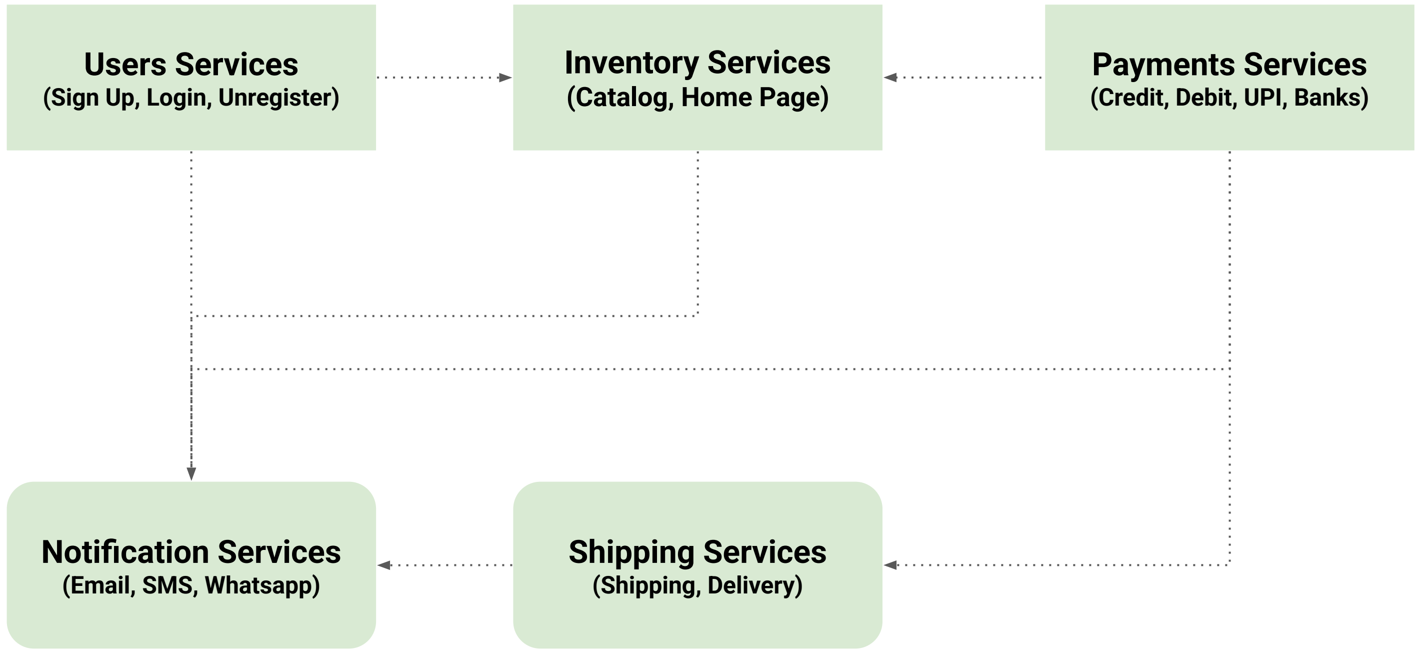
Which means those services have to work with each other (order service will need to talk to notification services whenever a purchase is done, order systems need to talk to payment services, etc)

Whenever two systems talk to each other - there are myriads of challenges that come in

Let’s understand those challenges and how event driven architecture solves those challenges

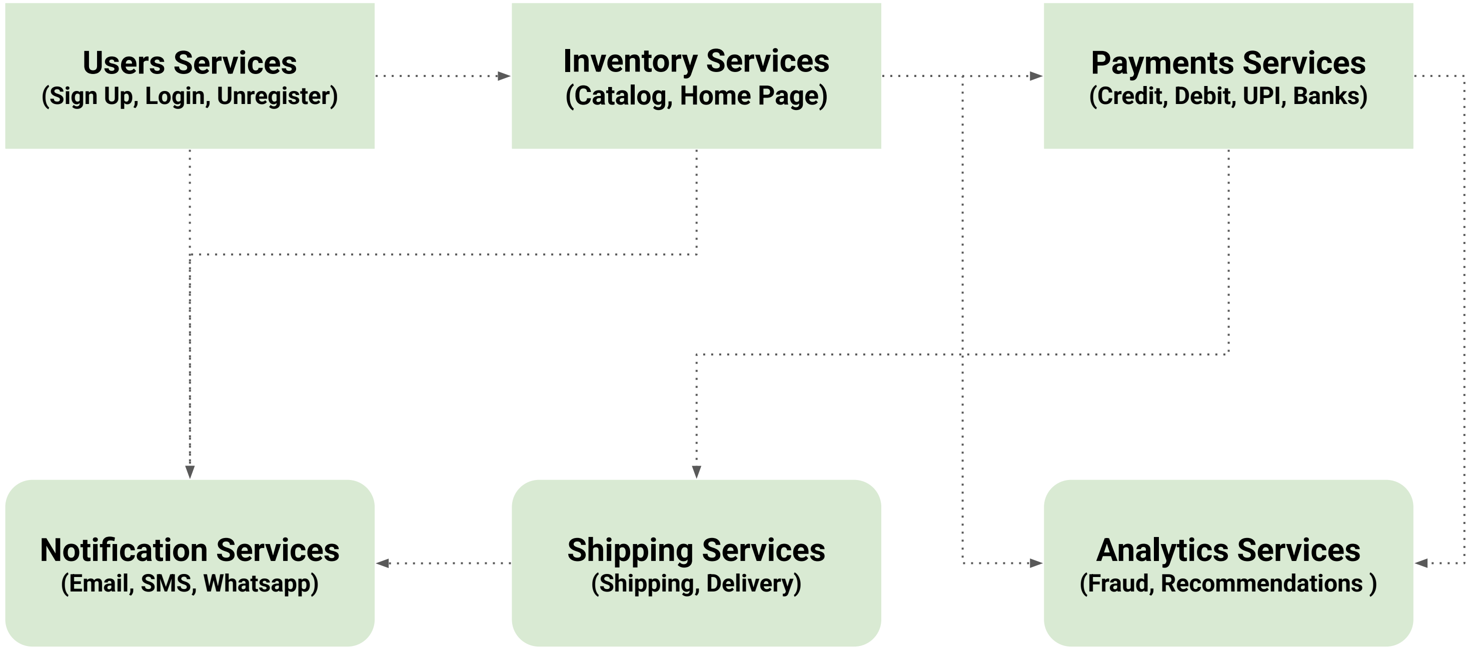
How Microservices Start: A typical E-Commerce

It starts small

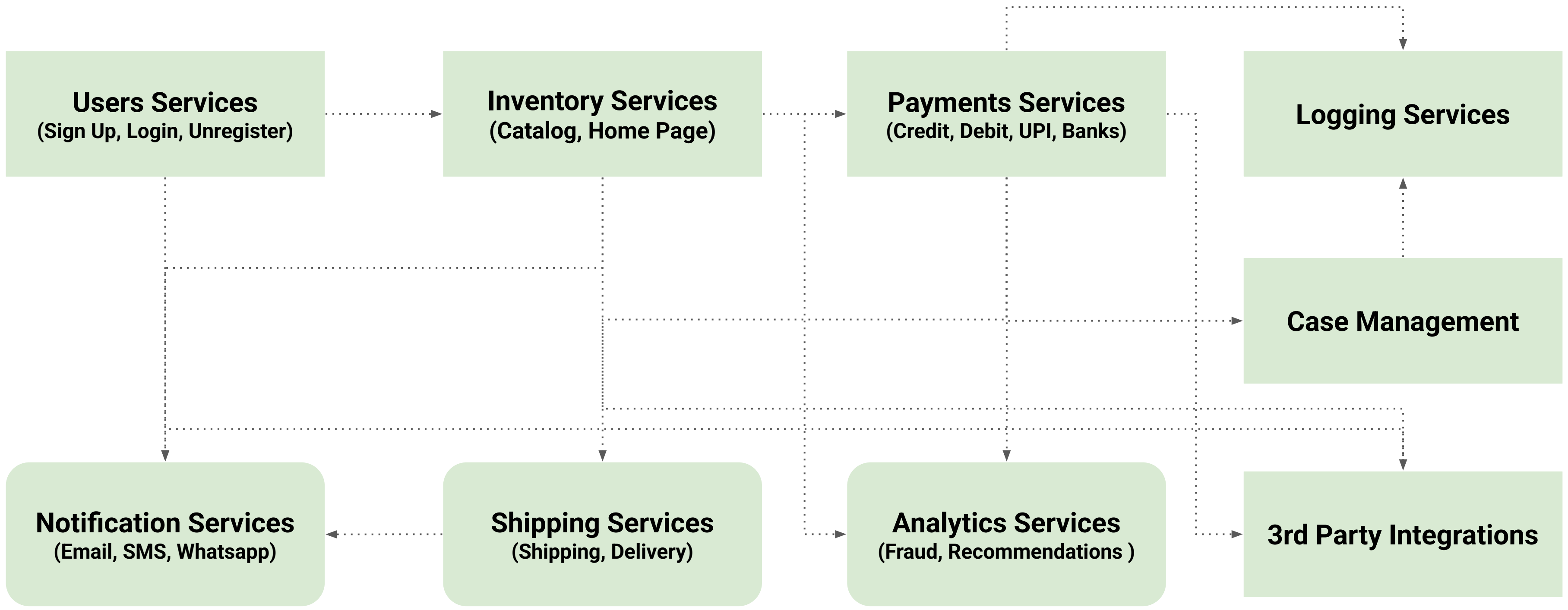


How Microservices Start: A typical E-Commerce

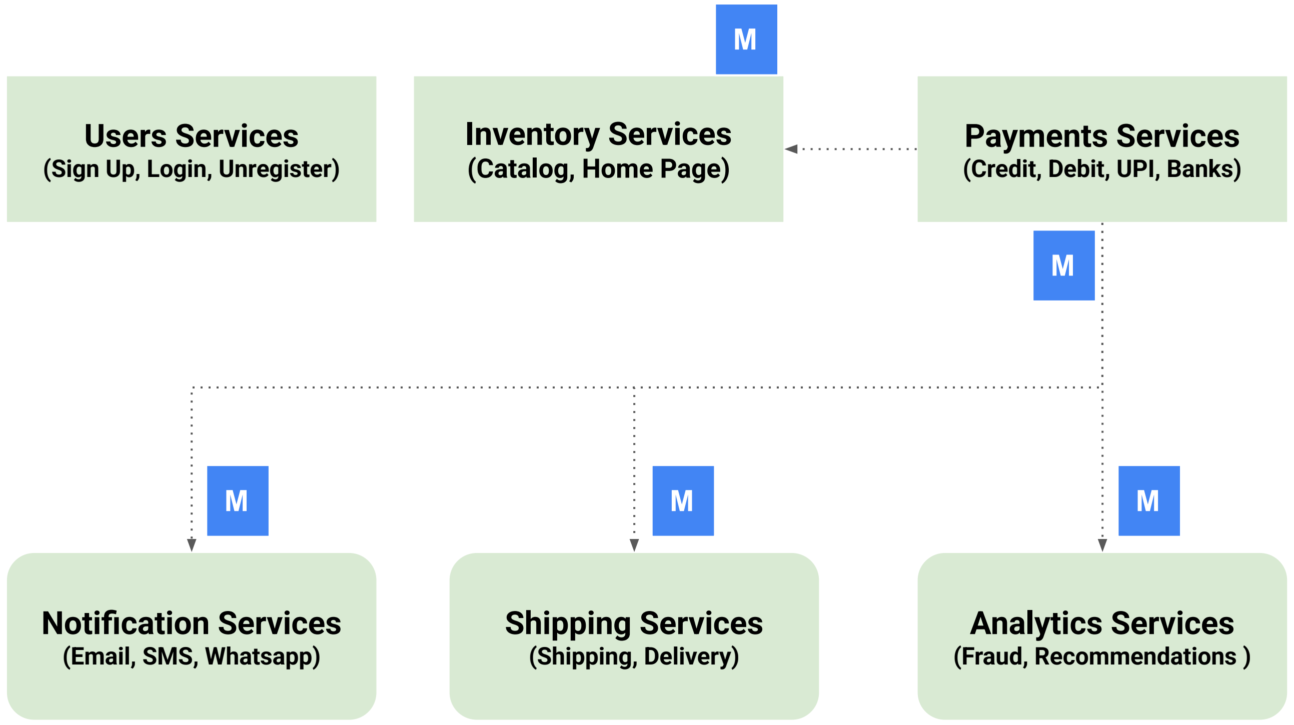
It starts small



How Microservices Start: A typical E-Commerce
It becomes really messy really soon



Problem 1: Scalability

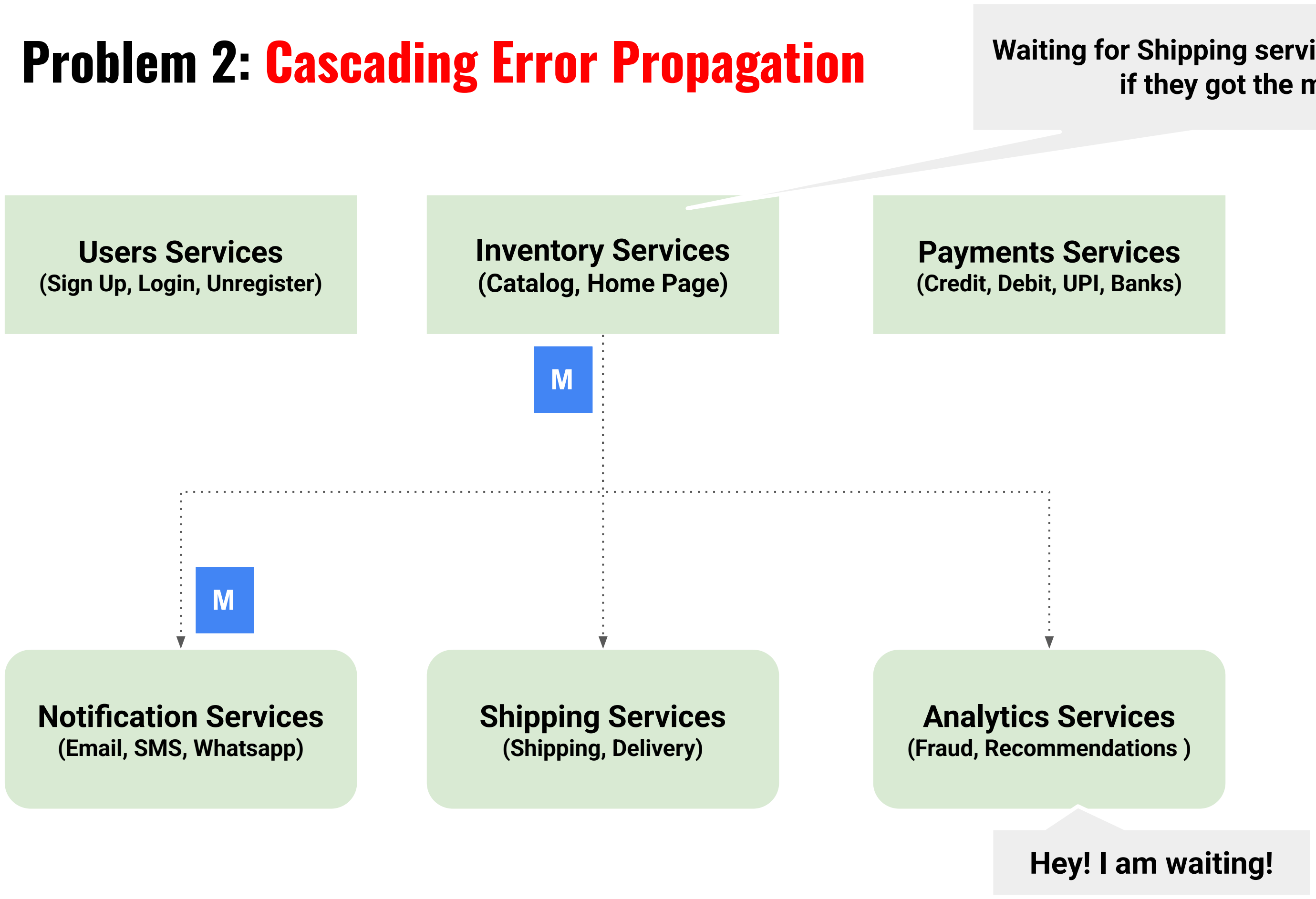


When a payments is successful, following services needs to be called:

- Inventory services to update inventory
- Shipping services: to schedule delivery
- Notification Services: to send order confirmation
- Analytics Service

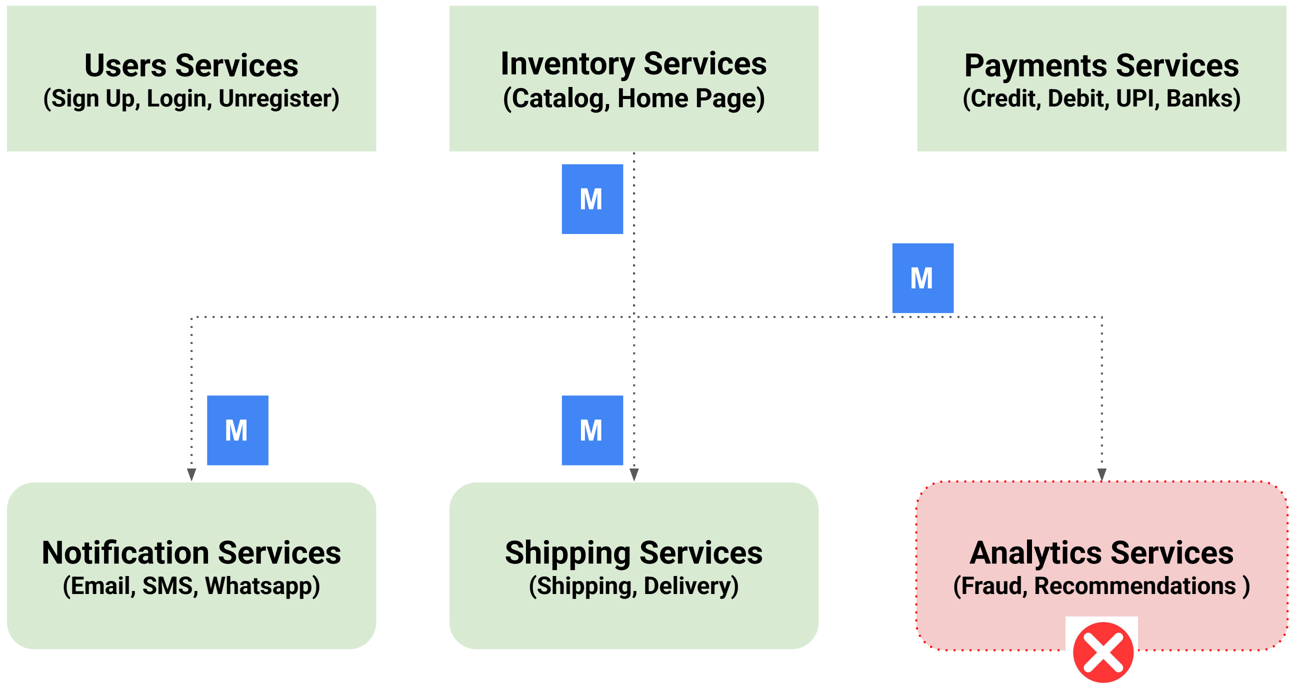
Sending same data to multiple system becomes the source application responsibility and it can have performance impacts if it has to send it to 100s of Target System

Problem 2: Cascading Error Propagation


Waiting for Shipping service to confirm if they got the msg
Users Services (Sign Up, Login, Unregister)
Inventory Services (Catalog, Home Page)
Payments Services (Credit, Debit, UPI, Banks)
M
M
Notification Services (Email, SMS, Whatsapp)
Shipping Services (Shipping, Delivery)
Analytics Services (Fraud, Recommendations)
Hey! I am waiting!

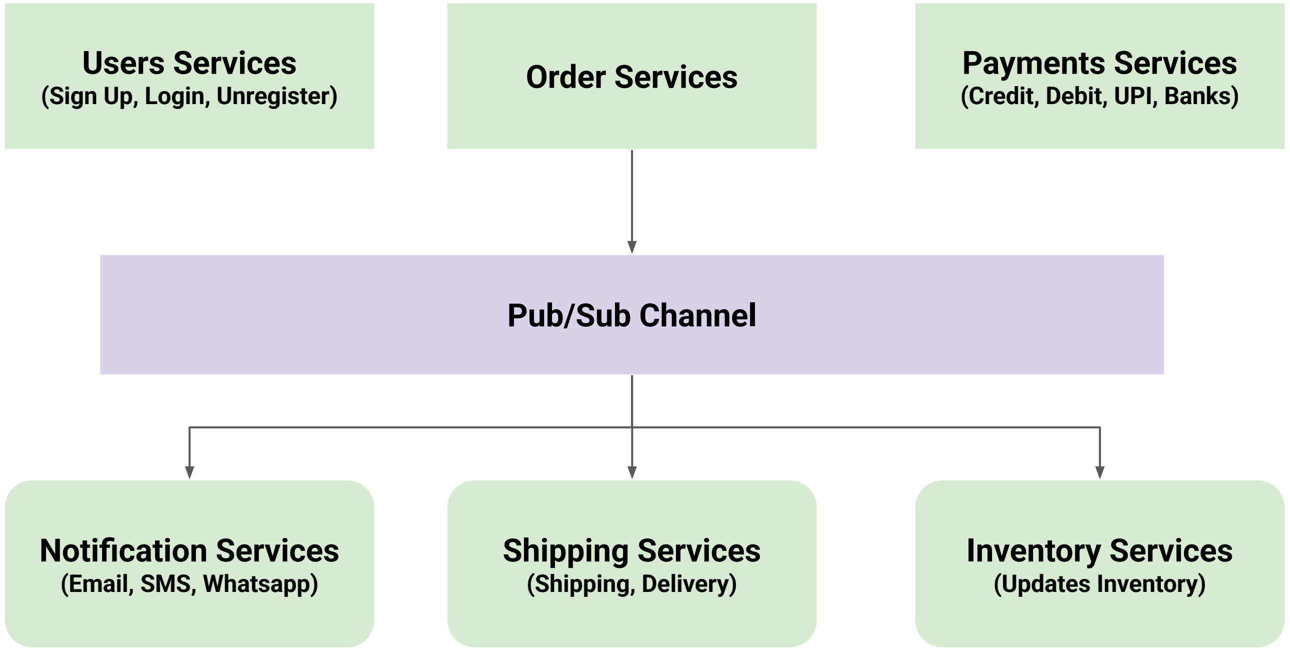
Since source and target system are talking to each other directly, any issue of target system can directly impact source system. Also, if target is slow to process or respond, Source also becomes slow

Problem 3: Durability



In the cases where Target system are down, the source systems cannot send the data to them and there will be data loss

Enter Event Driven APIs



Scalability: increases as source system does not have to send message to each system individually

Cascading: source and target do not talk directly, so errors in target system cannot impact source systems

Durability: even if target system goes down the channel will hold the message for sometime

The Instagram Problem

Imagine for a moment the scale at which Instagram works. How many **likes and comments** might the Instagram server be getting per second?

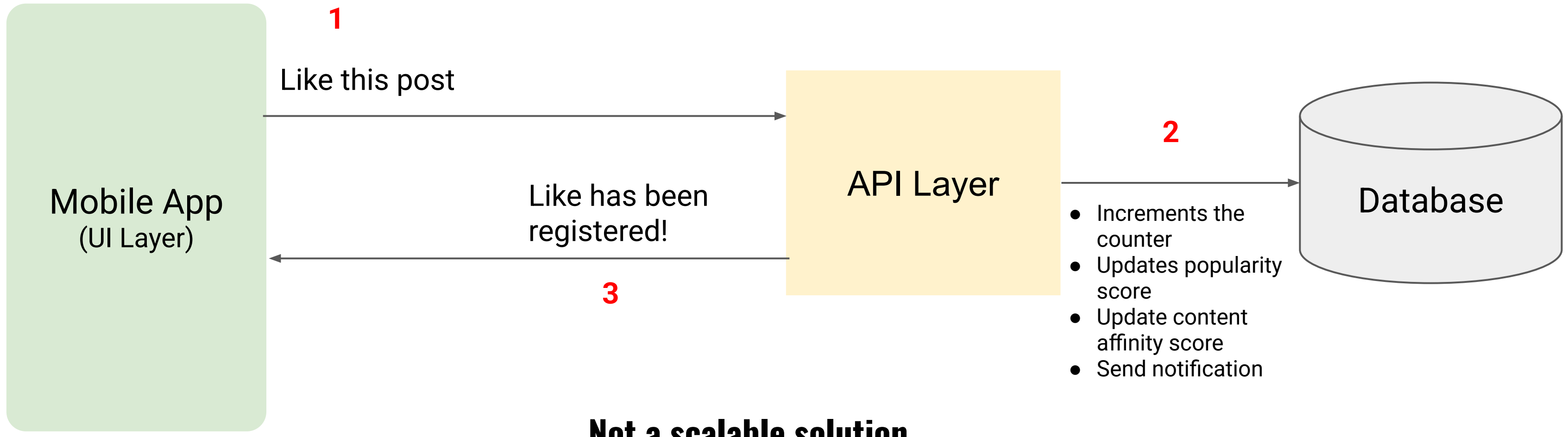
At such a high scale - **request/response model** starts showing its **limit** and we go for event driven APIs

Not only the high response time degrades UX, it also ties up your server process. Instead you want quick response and server to be free for processing more requests

The mobile or UI when it receives a like request - immediately shows the like registered on the UI while submitting the request to the API

API is programed to get the request and put it in a queue. Another process runs to process the messages put in the queue

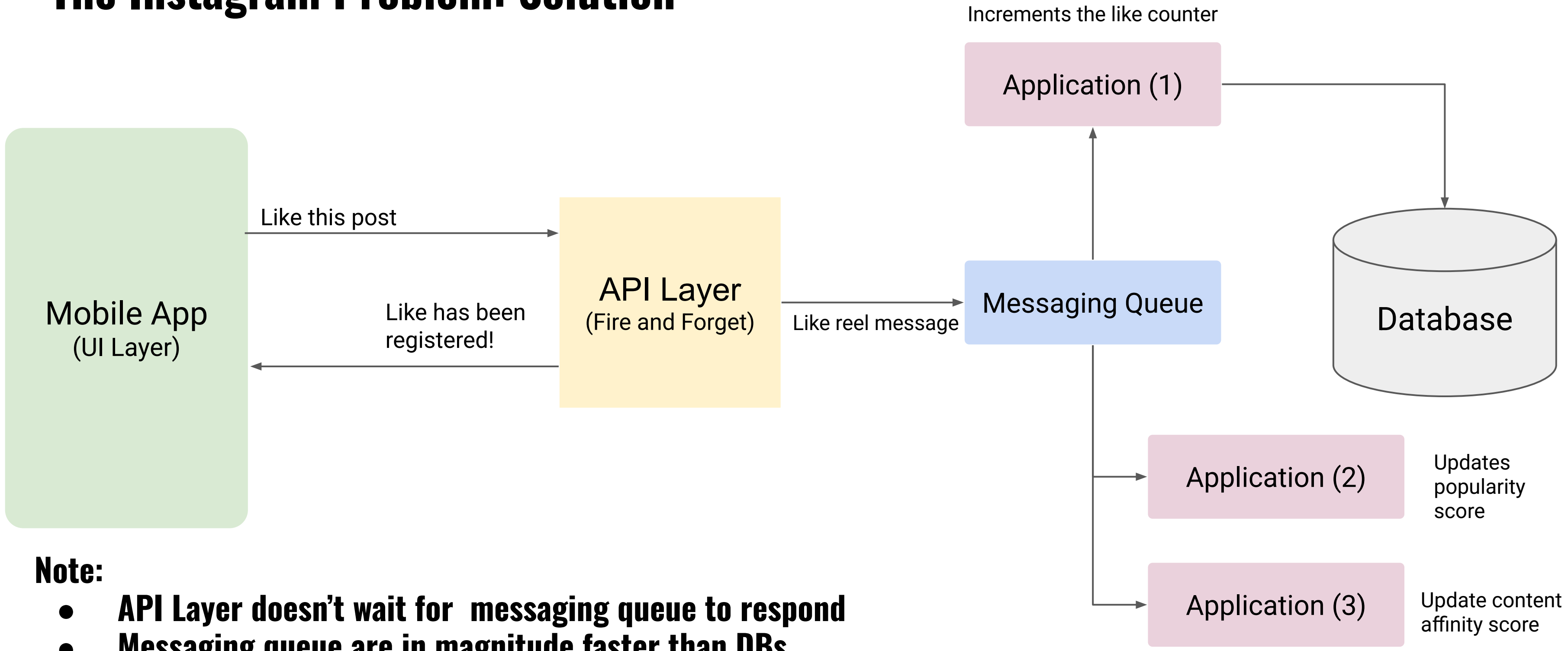
The Instagram Problem



- Increments the counter
- Updates popularity score
- Update content affinity score
- Send notification

Not a scalable solution

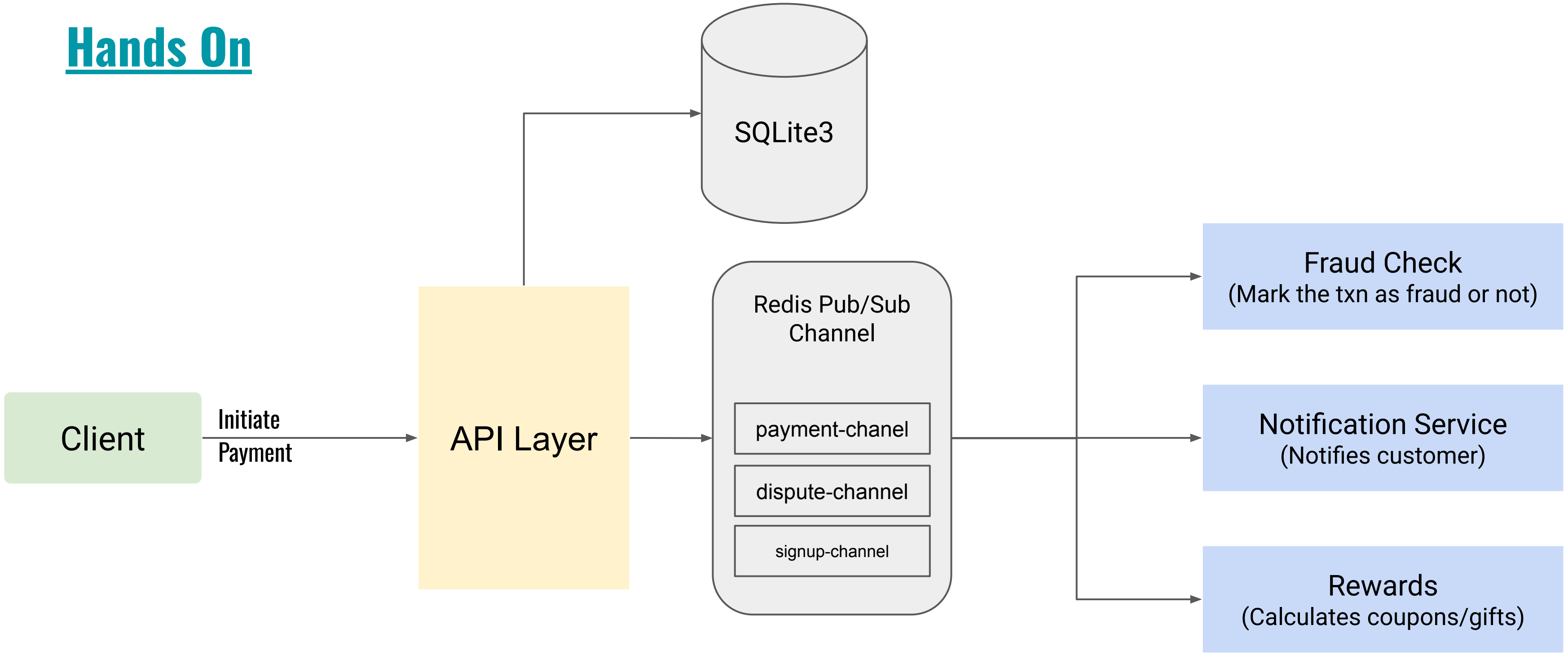
The Instagram Problem: Solution



Note:

- **API Layer doesn't wait for messaging queue to respond**
- **Messaging queue are in magnitude faster than DBs**

Hands On


SQLite3
Client
Initiate Payment
API Layer
Redis Pub/Sub Channel
payment-chanel
dispute-channel
signup-channel
Fraud Check
(Mark the txn as fraud or not)
Notification Service
(Notifies customer)
Rewards
(Calculates coupons/gifts)

Hands On - Link

DIY

Update the Fraud Detection listener in a way so that it updates the transaction status as “failed” when it detects “fraud” transaction and updates it as “success” otherwise

Another Application of Event Driven: Read Ahead Architecture

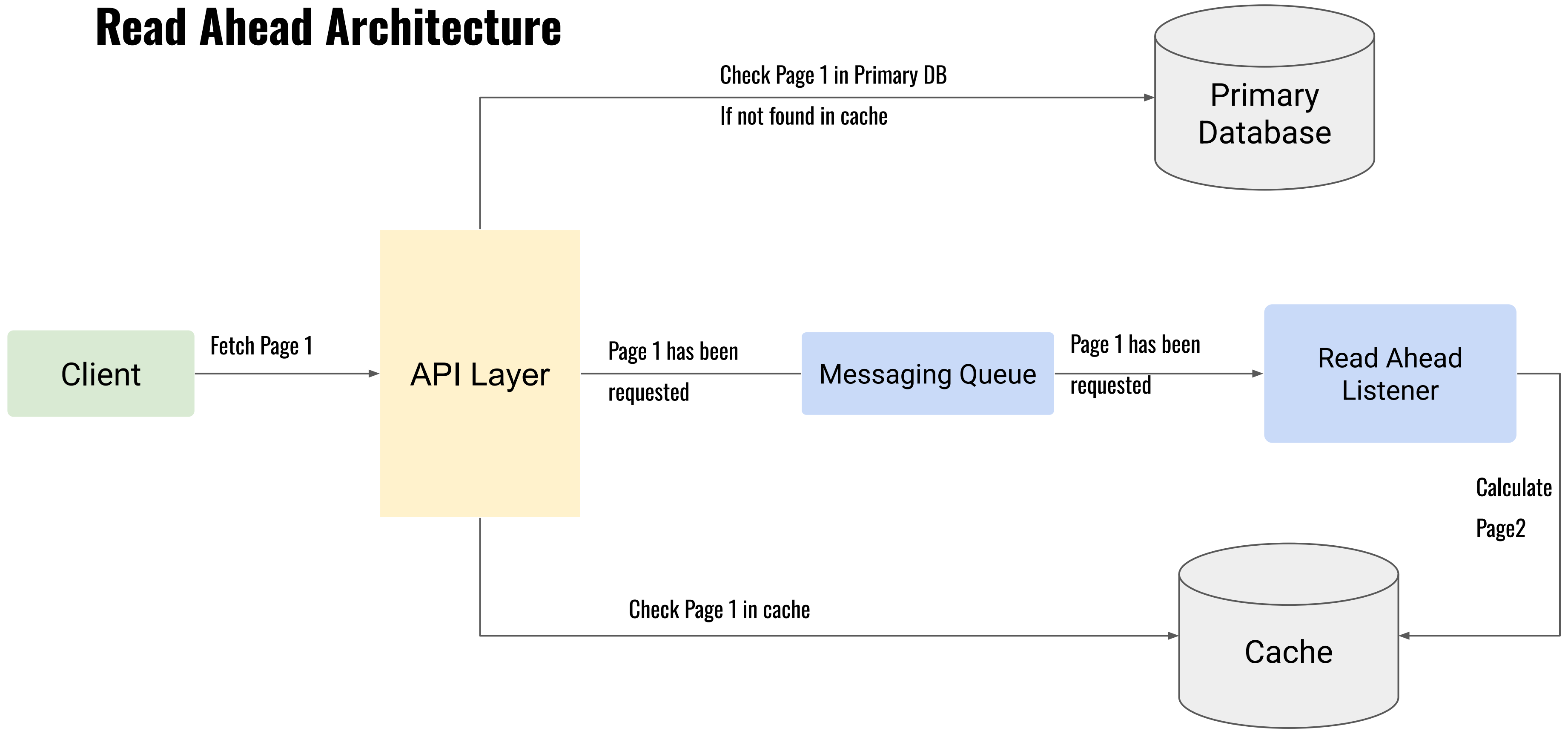
Imagine you have an API that need pagination implementation where a user can keep asking for new page

If we calculate the list of resource for each page when the request is made - it can be a little slow

Event driven APIs help us calculate before hand (read ahead) and keep the next page in cache

This helps in substantial improvement in response times

Read Ahead Architecture


Check Page 1 in Primary DB
If not found in cache
Primary Database
Client
Fetch Page 1
API Layer
Page 1 has been requested
Messaging Queue
Page 1 has been requested
Read Ahead Listener
Calculate Page2
Check Page 1 in cache
Cache