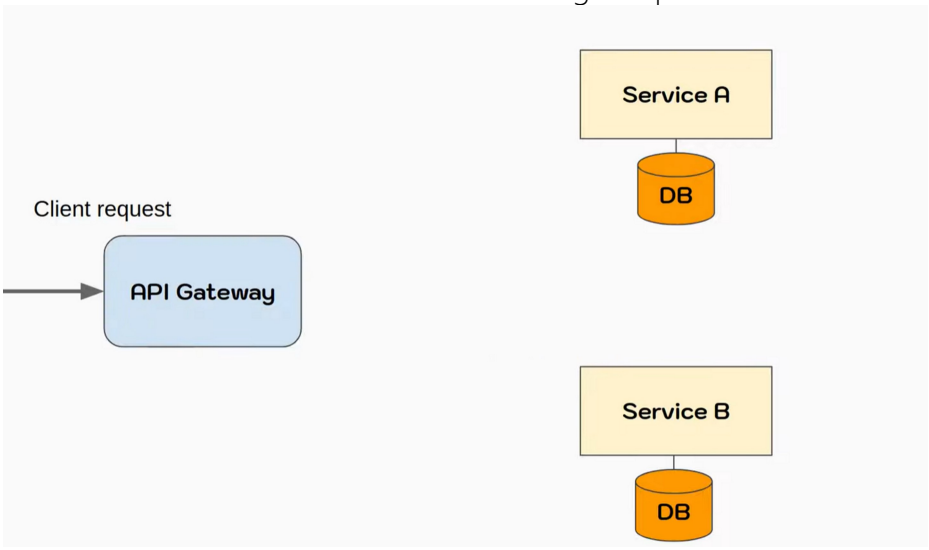
Domain Driven Design

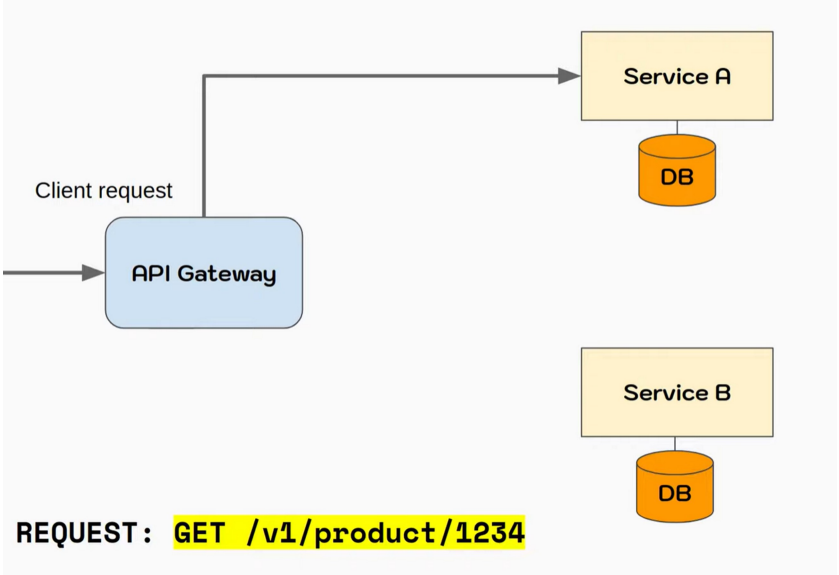
5 เมษายน 2565 13:38

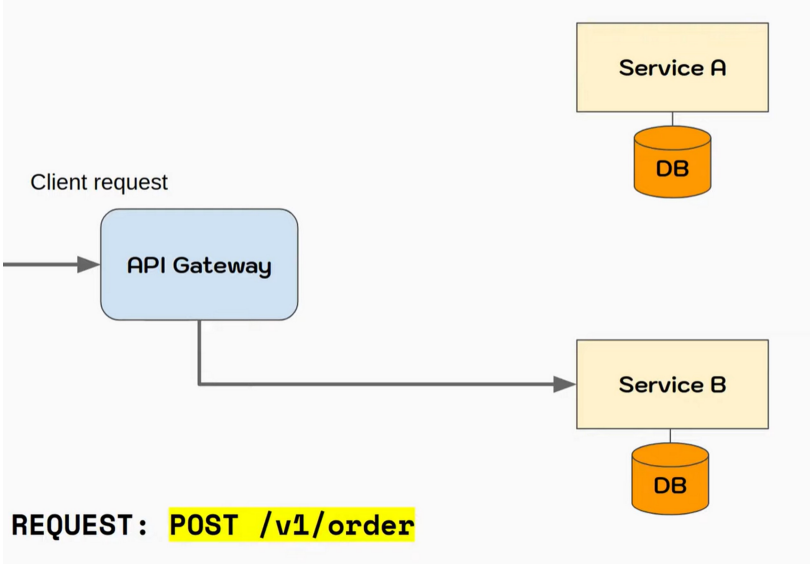
How Frontend talk to microservices

Problem: Microservice make all the service got separated

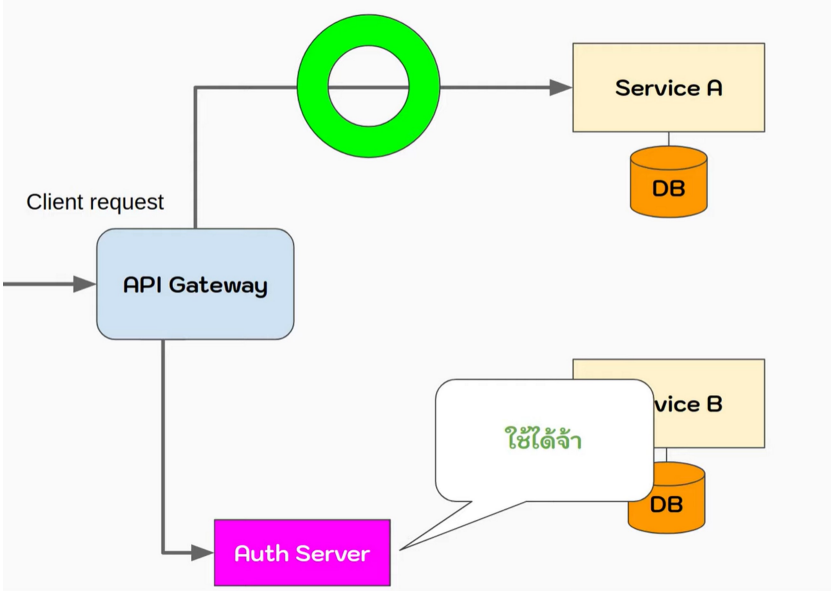


The easiest way to fix is to create an API gateway, which work as a proxy. Client can send request to the gateway, then the gateway will send the request to the related service. We need to program the gateway to know that which service is responsible for which endpoint.





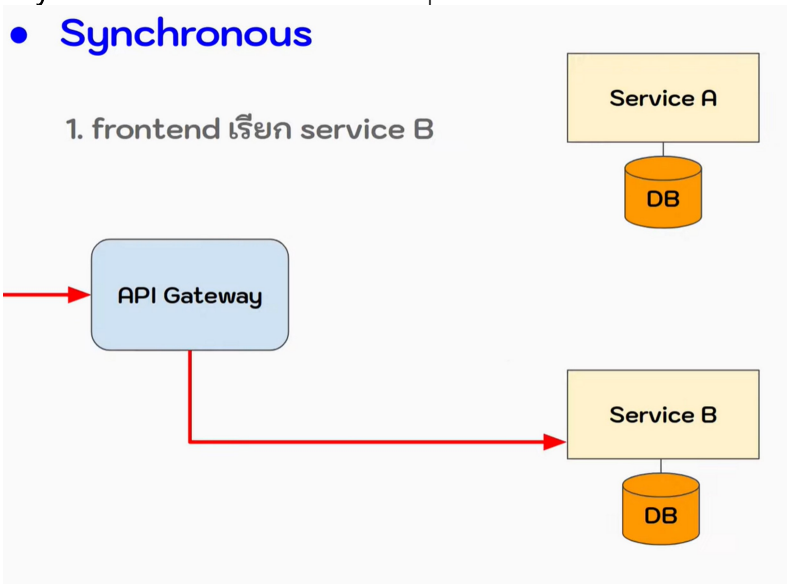
We can use just this but normally we will also have authentication. We will have authentication server, maybe another service on docker or third party server.
For the flow, when client send a request we will check for header authentication, it must have access token in the header. We will also have to check if the token is valid or not. Gateway can check token by itself with public key or send request to authentication server. After the validation pass gateway will allow client to reach services along with client info.

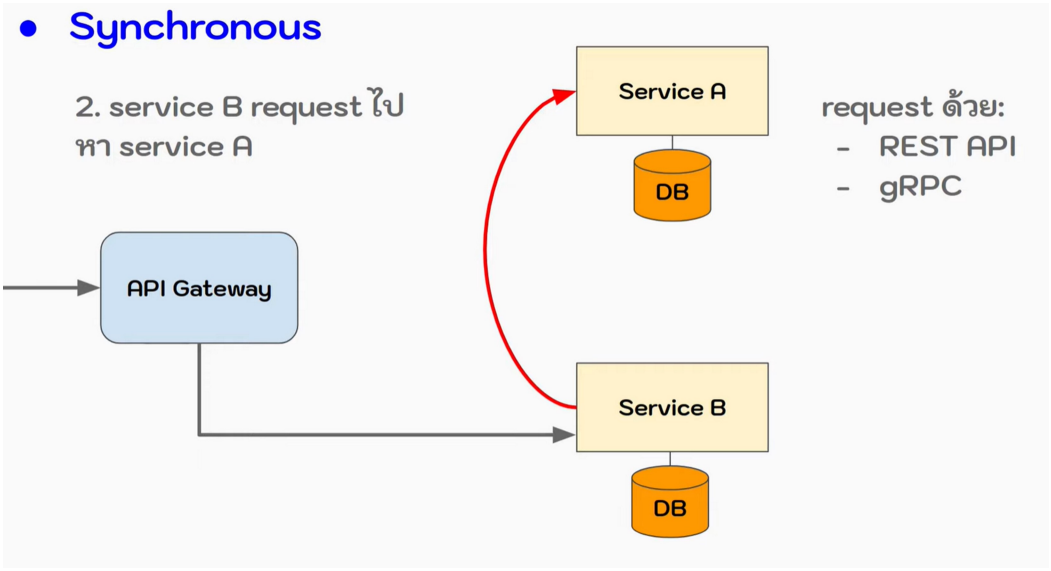


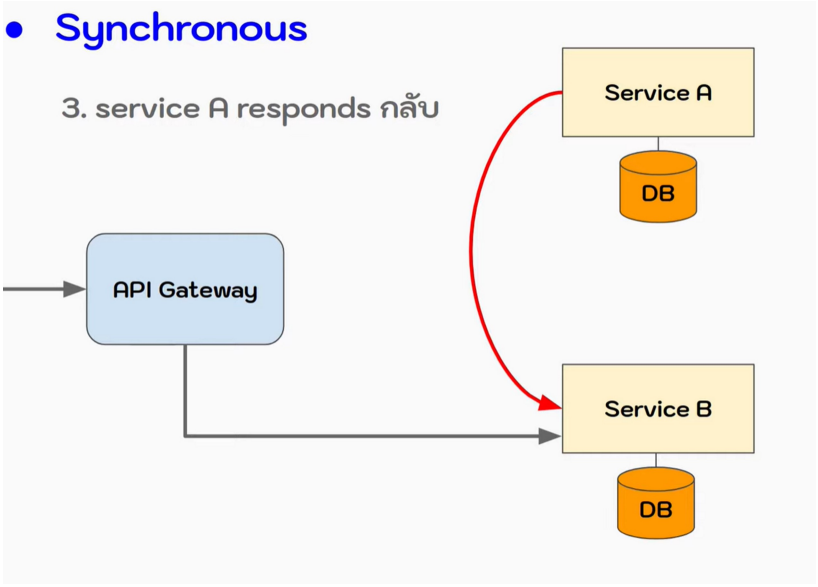
And refuse access otherwise.
Pros of this method is: we don't need to write authentication code for every microservices we use.

How Microservices talk to Microservices

Ideally when we have frontend request to one service, then the service response back, the end, easy as fuck. But actually no, service will use functionality of another service.
Problem: service need to talk to each other.
Synchronous - send a request, wait till finish, accept response
Asynchronous - not wait for response

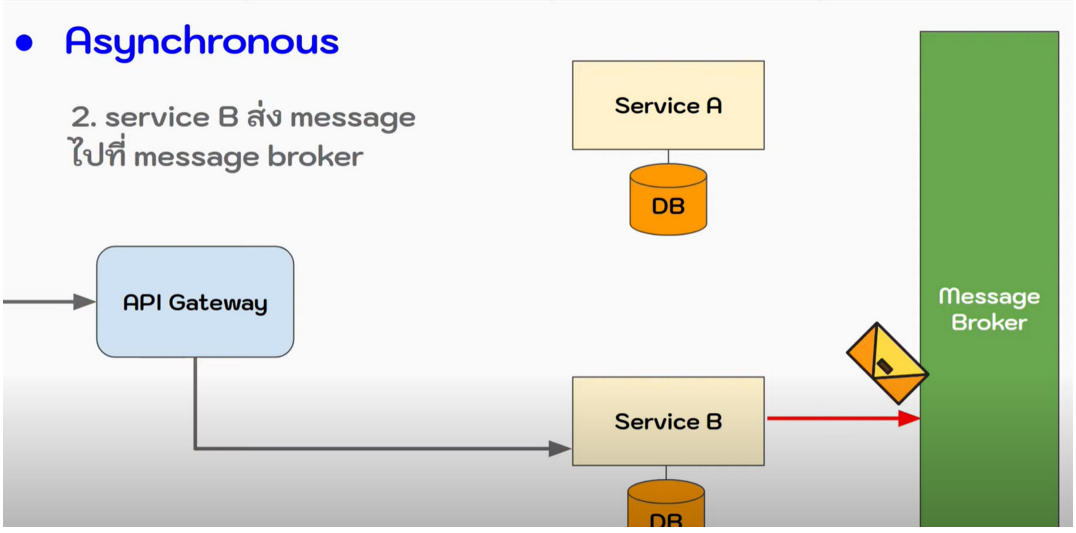


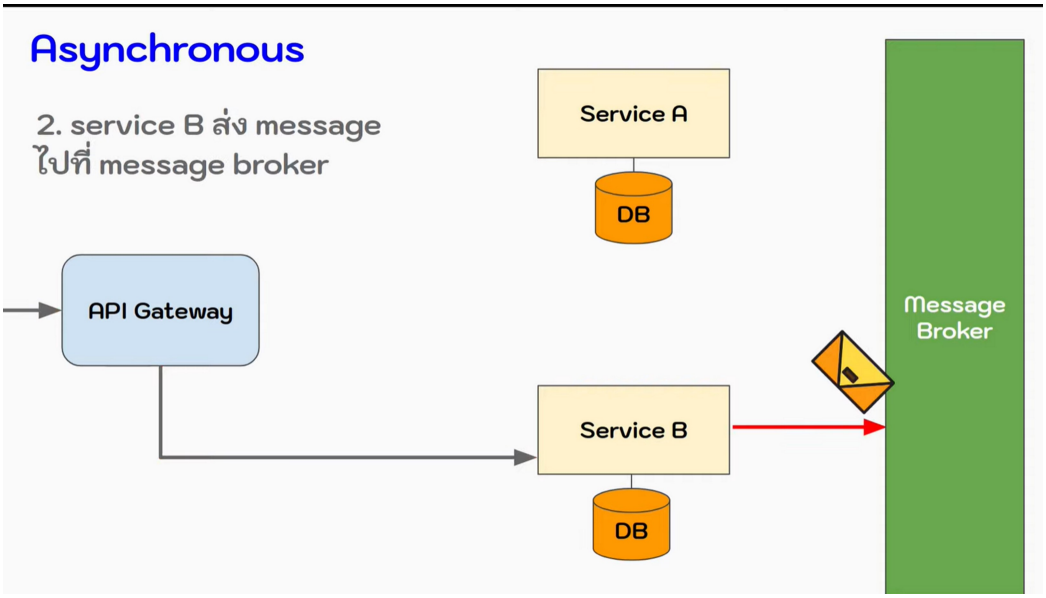


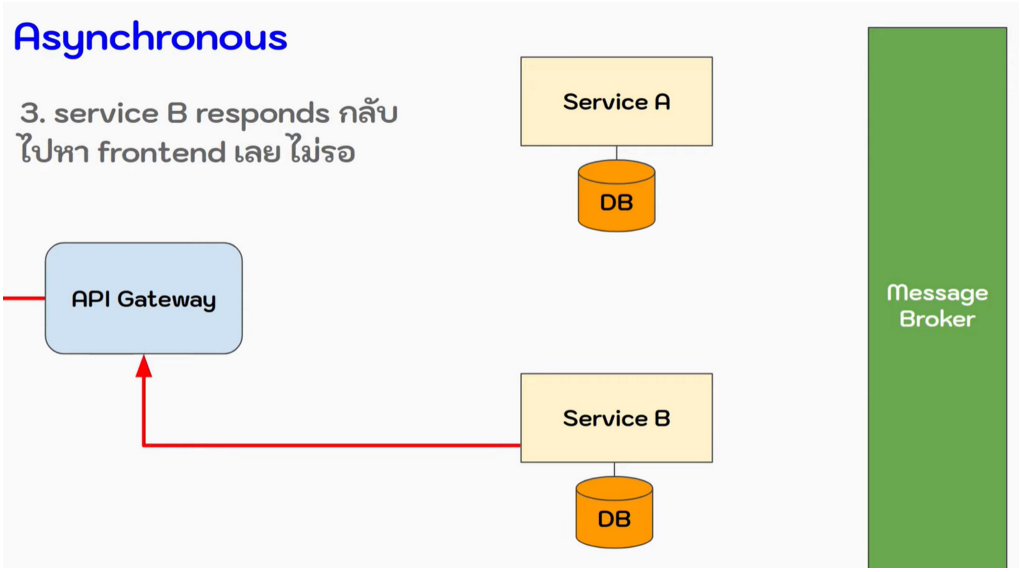


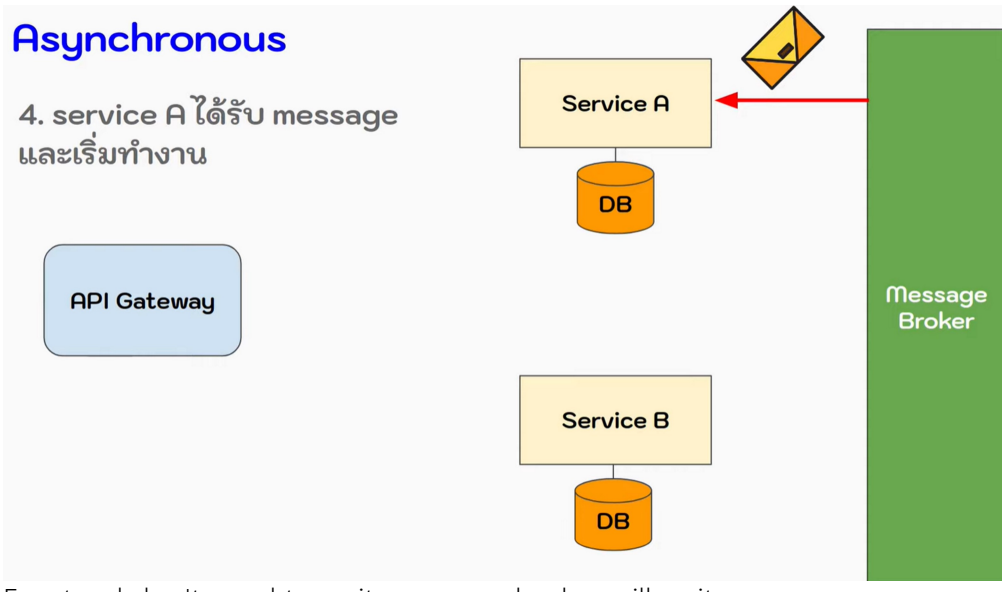
All of this frontend still waiting for us. When service B finish, send the results to frontend.
Cons:
frontend need to wait longer
Endpoint service slow or dead i.e. if service A just die what will service B do? Retry? Die? Send another request?

So to prevent these problem we use asynchronous
Use message broker; like a postbox, service throw mails in here 'send it to that service pls'

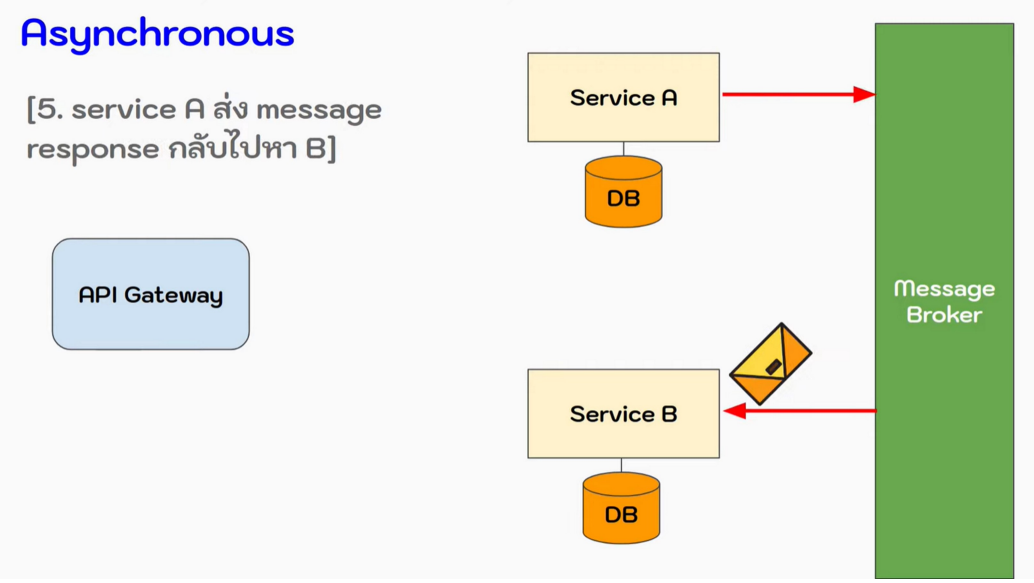




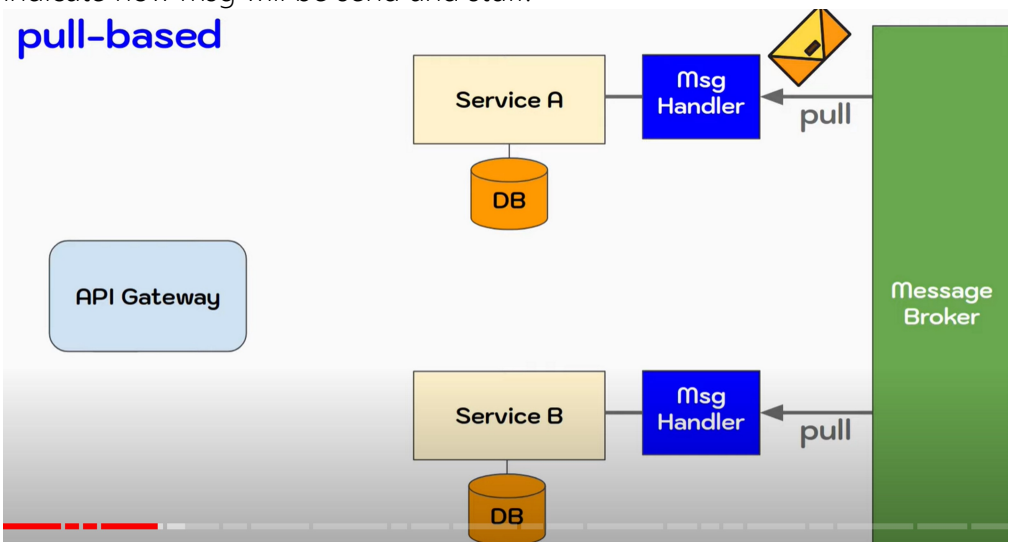




Frontend don't need to wait, message broker will wait.



Type of message broker
Push-based - shoot message to services, just open server and wait
Pull-based - each service has to query to check if there's a message or not
The above example is push-based, we need to do some configuration in the broker to indicate how msg will be send and stuff.



We might have a process or thread to query msg to run, easy to scale

ตัวอย่าง Message Broker ในตลาด

RabbitMQ (push)

kafka (pull)

Google Cloud Pub/Sub (push&pull)

Logging & Monitoring

Problem:

Service is separated, not together

Complicated process

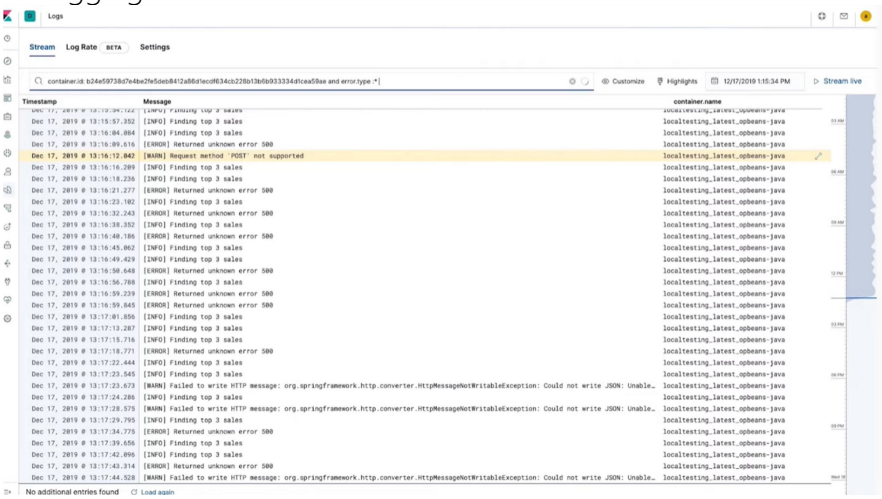
So need a tool to help gather log and matrix in our system.

What can be monitor in microservice?

1. Infrastructure monitoring

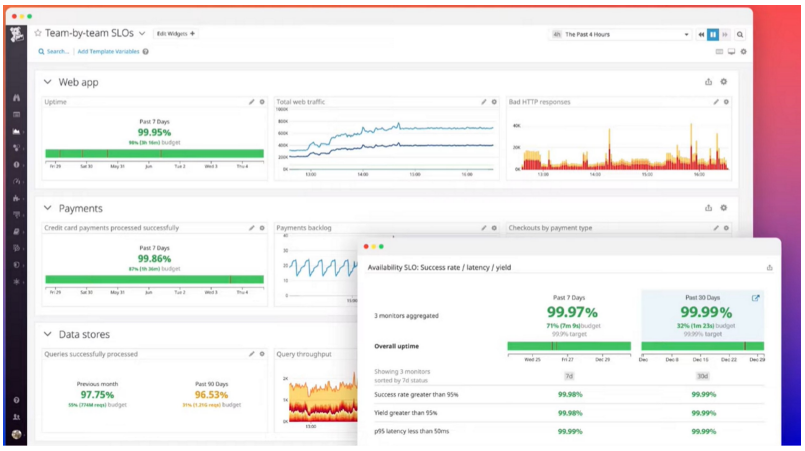
Monitor that each services/pods still healthy, the amount of container, % CPU RAM, bla bla bla

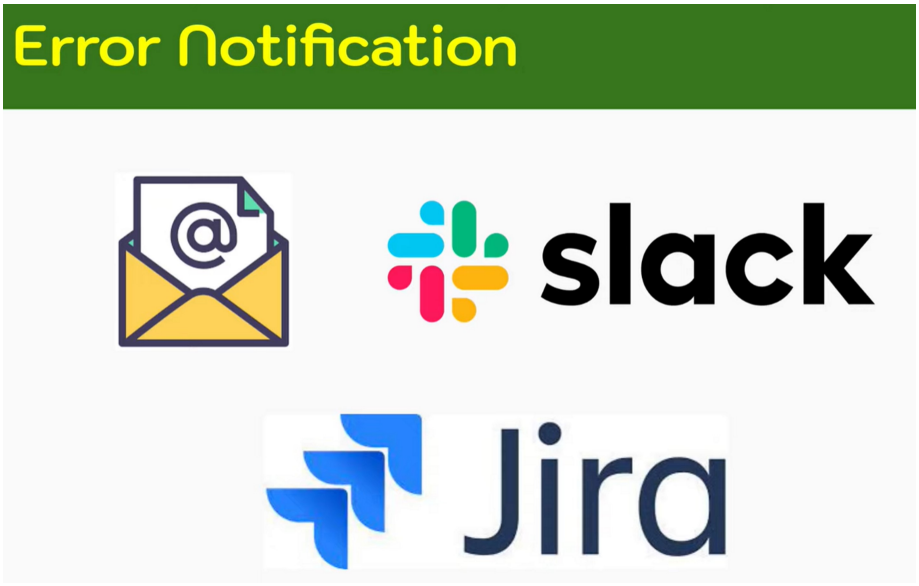
2. Logging



3. service-wise metrics

Like a dashboard that show metrics of interest



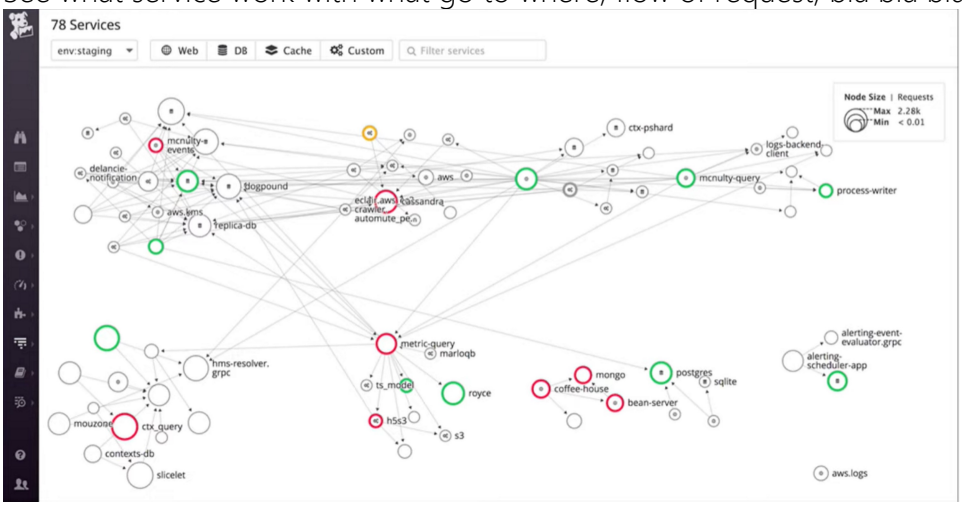


We also need noti along with dashboard

4. debug

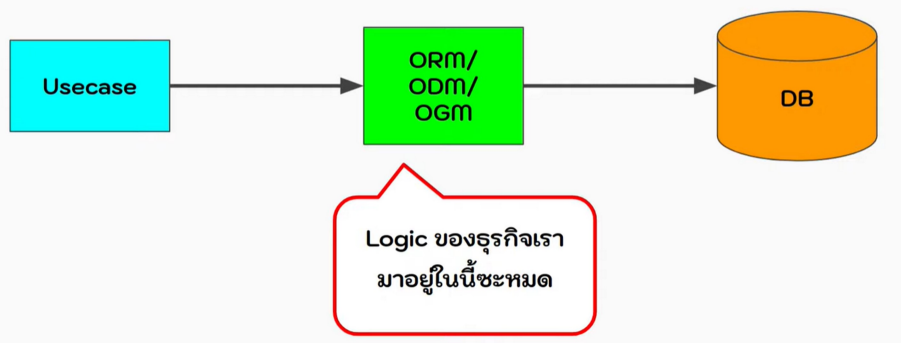
Service map

See what service work with what go to where, flow of request, bla bla bla



Data Access/Manipulation

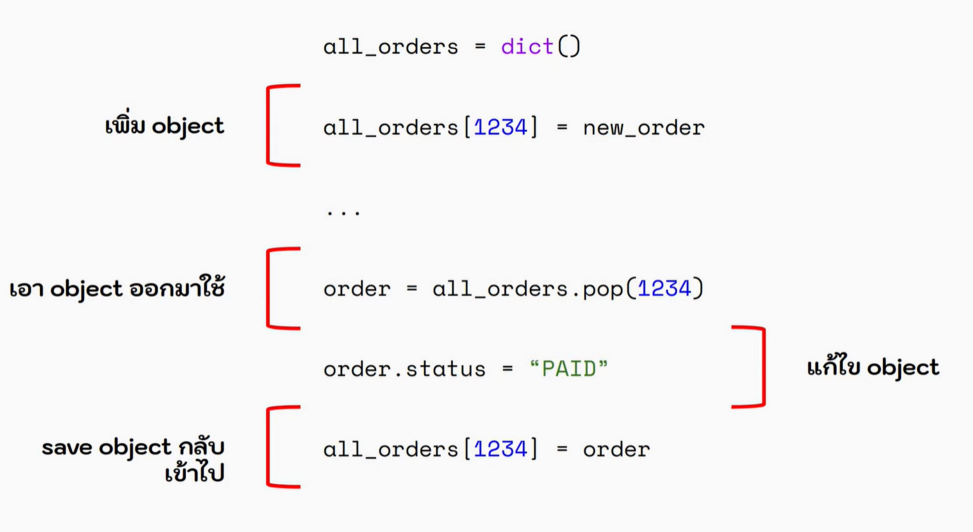
When our service interact with database we use tools like ORM/ SQL but the problem is: Logic of our business will all be in there. So it will be hard to understand or debug (programming language are more easy to understand than ORM or SQL)



So How do we implement logic in a form of programming language?
Do operations in a form of transaction

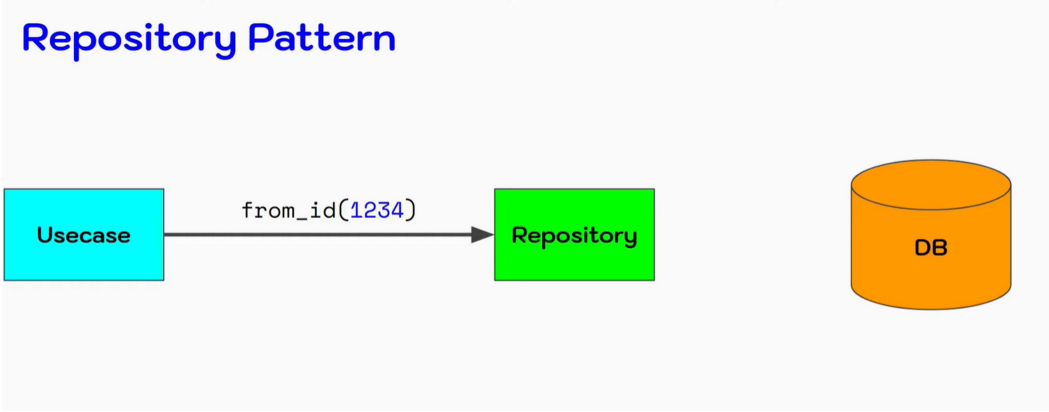
SOL 1: Repository Pattern

Similar to how you use python dict, we will database as just dict only usage is for collecting data, no edit or fixing data

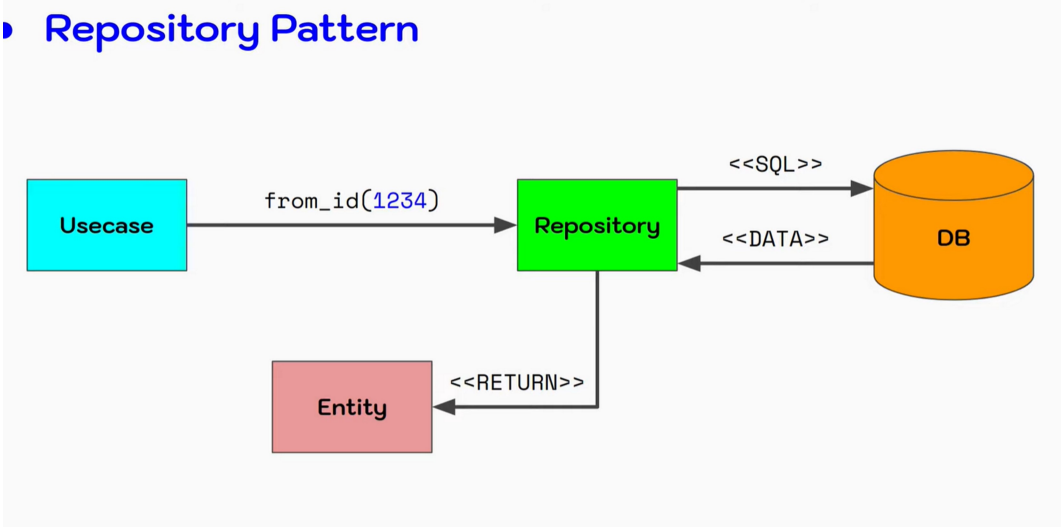


Repository will be a class we created.

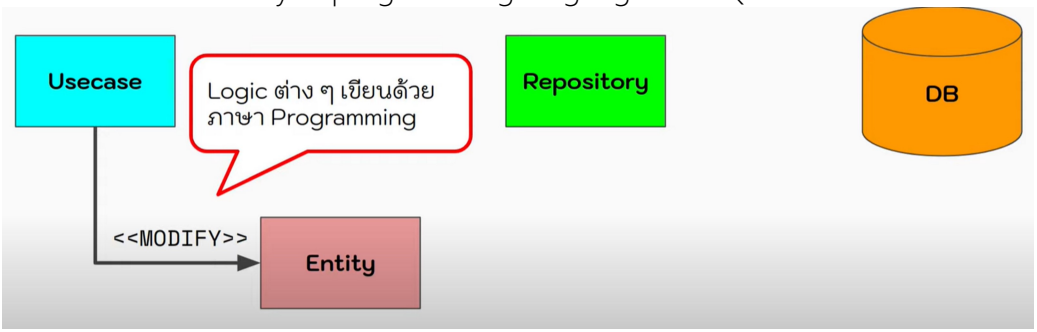
When we want to edit data we will ask for data from repository.

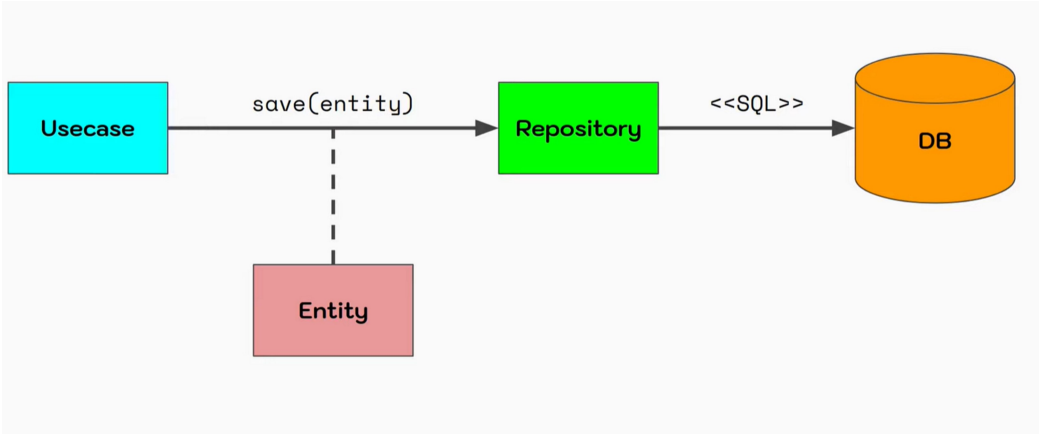


Repository will then query the object from the database (maybe implement as SQL or ORM)



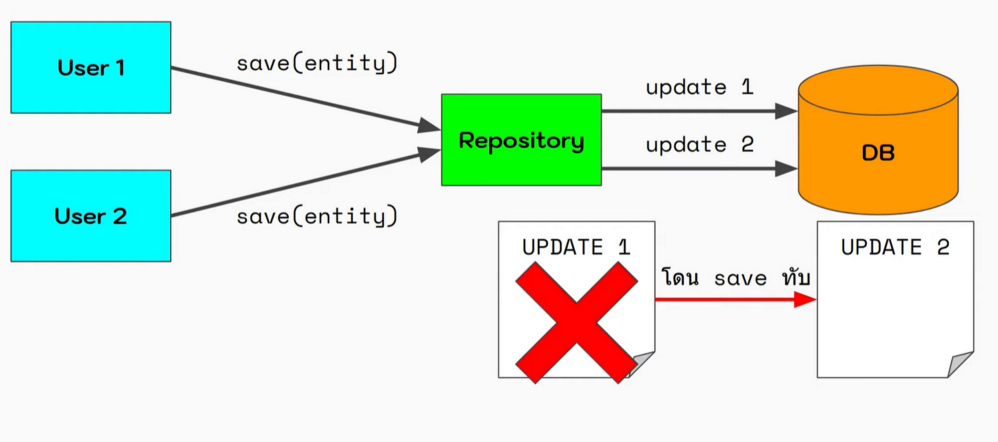
We will edit this entity in programming language not SQL n'stuff.





Save new data through repository

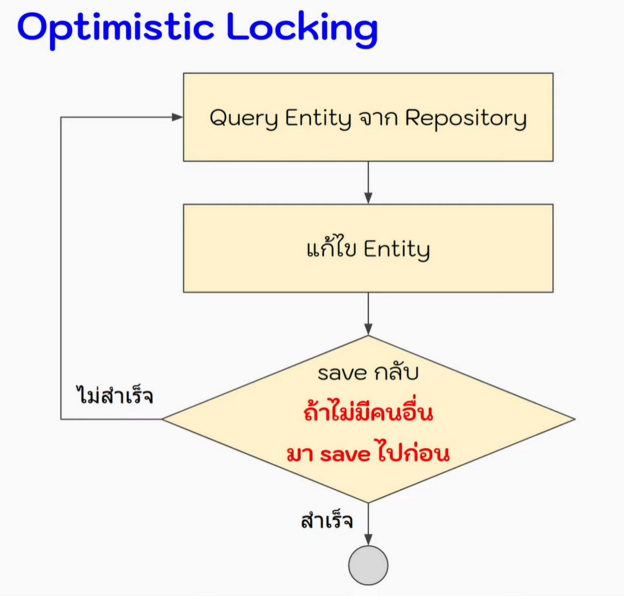
But there's a problem: Like when 2 users editing the same entity



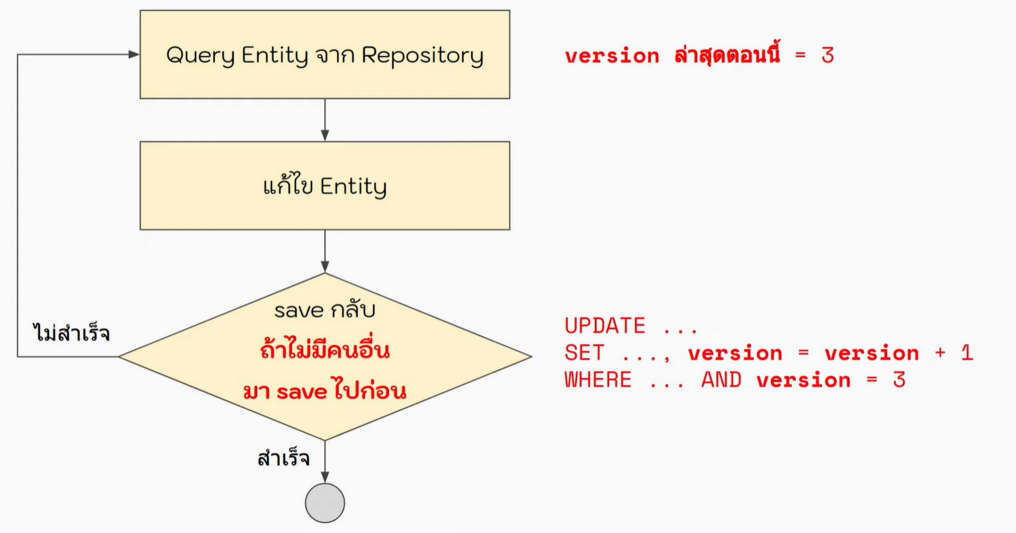
So we turn it into a transaction

Optimistic Locking

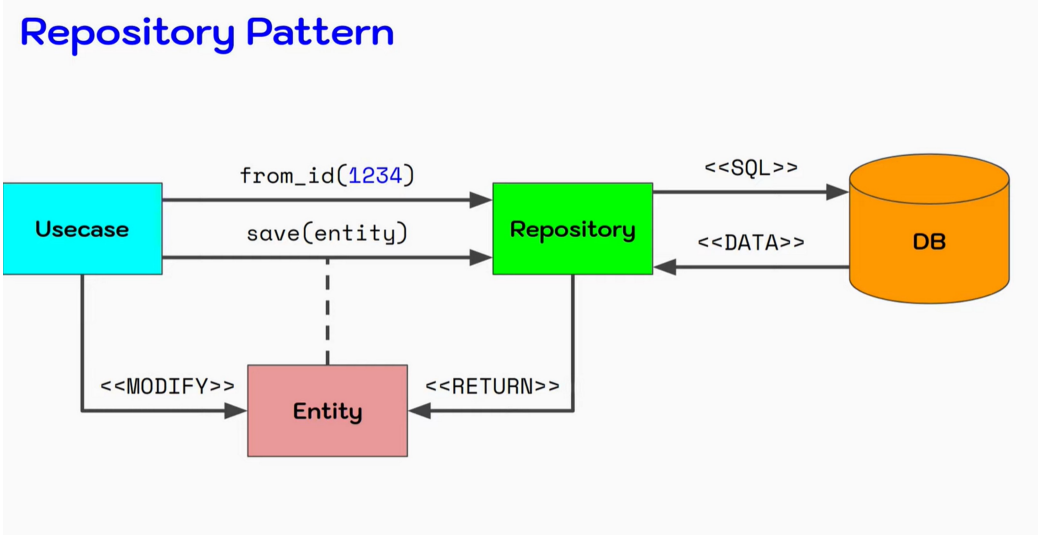
Query entity from repository, edit it, save it back through the repository, but now we have a condition; during the time of editing there must be no user save said entity before us. So we will not overwrite anyone.



The important part is how will the database know when there's person save before we do?



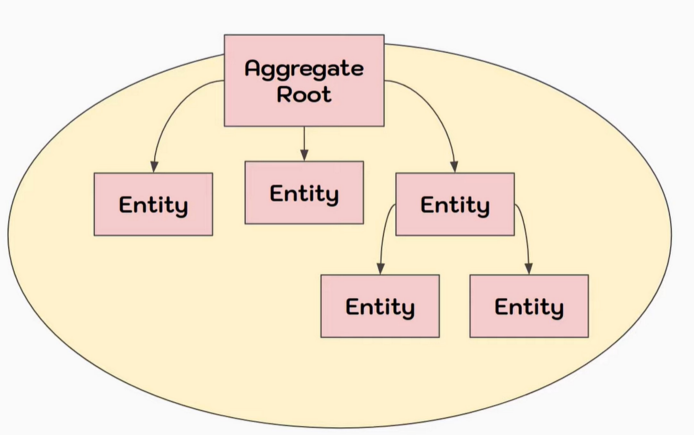
The easiest way is to add attribute version which will be implemented every update.



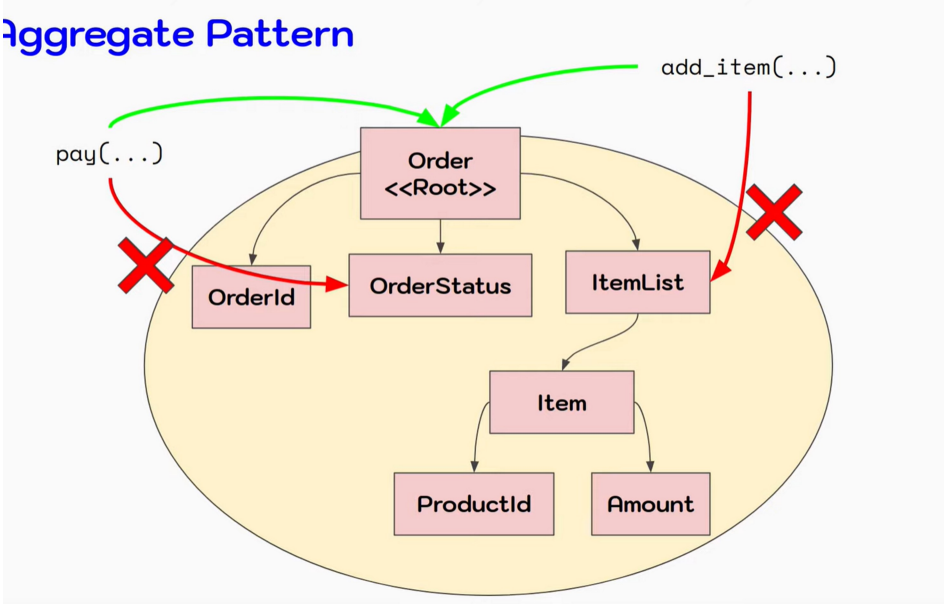
So repository pattern will help us implement logic in programming language, make code more readable, easier to fix and test, debug
Then how do we join data between 2 repository?
Well, the video said 'you don't actually want to join it, think again' or just use aggregate pattern.
When there's many user using the same entity (high concurrency) will optimistic locking still work? Also see aggregate pattern

Aggregate Pattern

Things you call 'Aggregate' is a group of entity or obj that related in a group as a unit of transaction, meaning changes that happen with aggregate will happen inside itself as 1 transaction.



To use aggregate we have condition that we need to call method through aggregate root only, cannot call or edit from other entity. Because to control integrity of data.



ไม่ใช้ aggregate

```
order = all_orders.from_id(5555)

if order.status == "PAID":
  raise OrderAlreadyPaidException()
else:
  order.items.append(new_item)

all_orders.save(order)
```

Use getter, but have to repeat this If code many places

ไม่ใช้ aggregate

```
order = all_orders.from_id(5555)

if order.status == "PAID":
  raise OrderAlreadyPaidException()
elif order.status == "CANCELLED":
  raise OrderAlreadyCancelledException()
else:
  order.items.append(new_item)

all_orders.save(order)
```

Like need to add this shit everywhere.

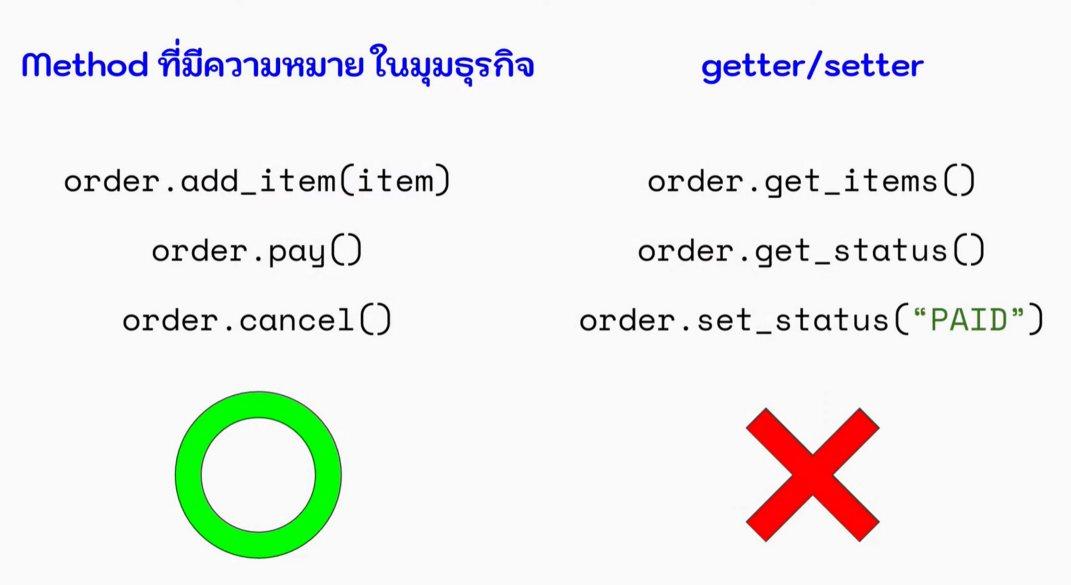
นิยาม aggregate

```
class Order(Aggregate):
  ...
  def add_item(self, item):
    if self.status == "PAID":
      raise OrderAlreadyPaidException()
    elif self.status == "CANCELLED":
      raise OrderAlreadyCancelledException()
    else:
      self.items.append(new_item)
```

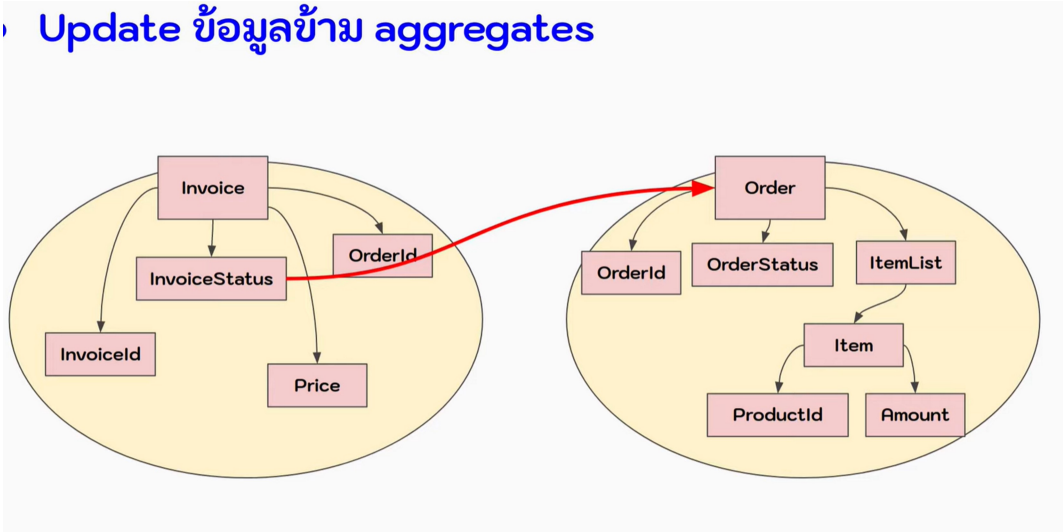
Make a method.

ใช้ aggregate

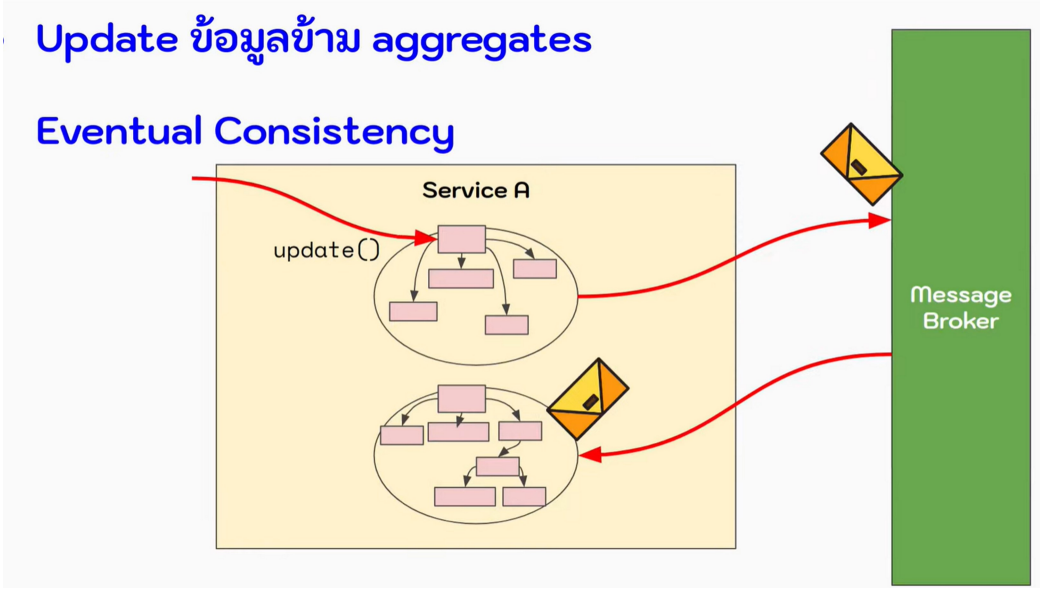
```
order = all_orders.from_id(5555)

order.add_item(new_item)

all_orders.save(order)
```



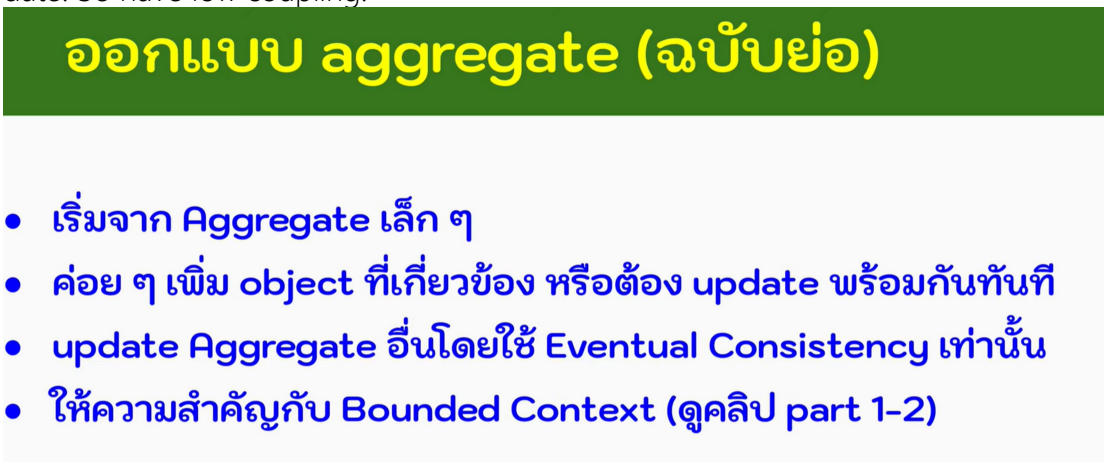
Good aggregate method must be a unit of transaction; things that need to be done together and have meaning in business term, not like getter setter.



But aggregate is a unit of transaction which mean only one update can happen per transaction. So how to join data?



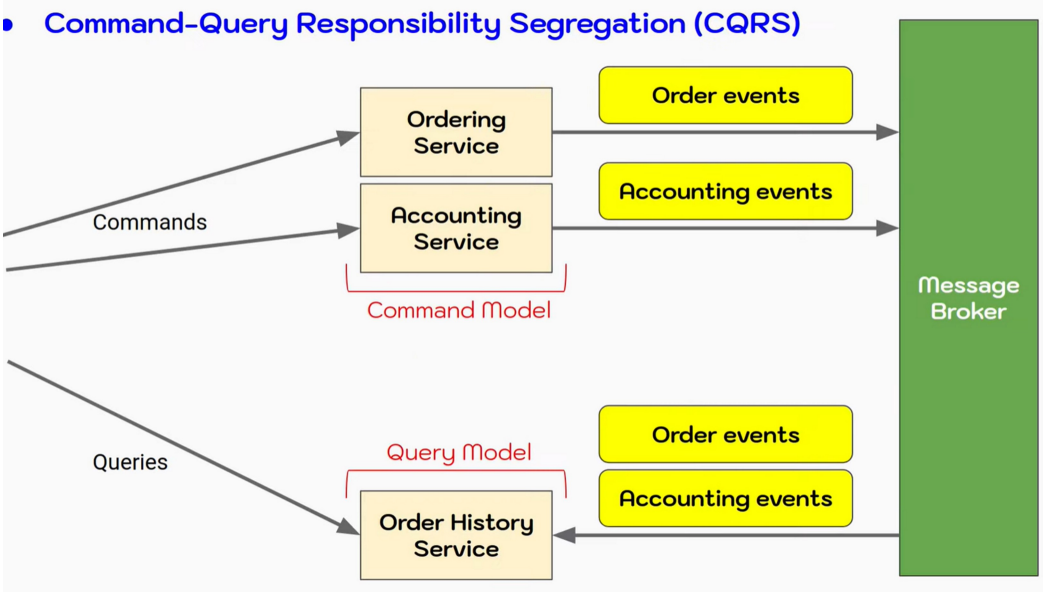
Use event to trigger update asynchronously. Eventual consistency, eventually it will be up-to-date. So have low coupling.



Also optimistic locking will still work depending on your aggregate design.

Bonus: CQRS pattern: create services with different jobs, have service which is a command model and another a query model. It can be use to fix the problem of data separation,

separate database. In this pattern user will perform actions/commands via command model only, then it will publish event to msg broker which have query model reading said event. Then it query model will project it onto it's own database so frontend can use it easily. Event that query model accept can be from multiple services and join here, but usage for query only, update will be done in command modem



Event-based Communication

```
Submit Order
order = all_orders.from_id(5555)

order.submit()

all_orders.save(order)  # update ข้อมูลใน db
```

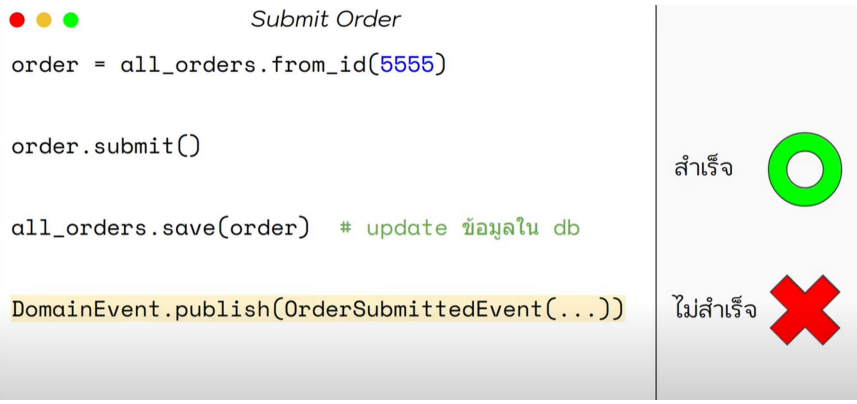
I have a code, so I want it so that when there's an order submit event must also be publish. How can I do that?

```
Submit Order
DomainEvent.publish(OrderSubmittedEvent(...))

order = all_orders.from_id(5555)

order.submit()

all_orders.save(order)  # update ข้อมูลใน db
```

สำเร็จ

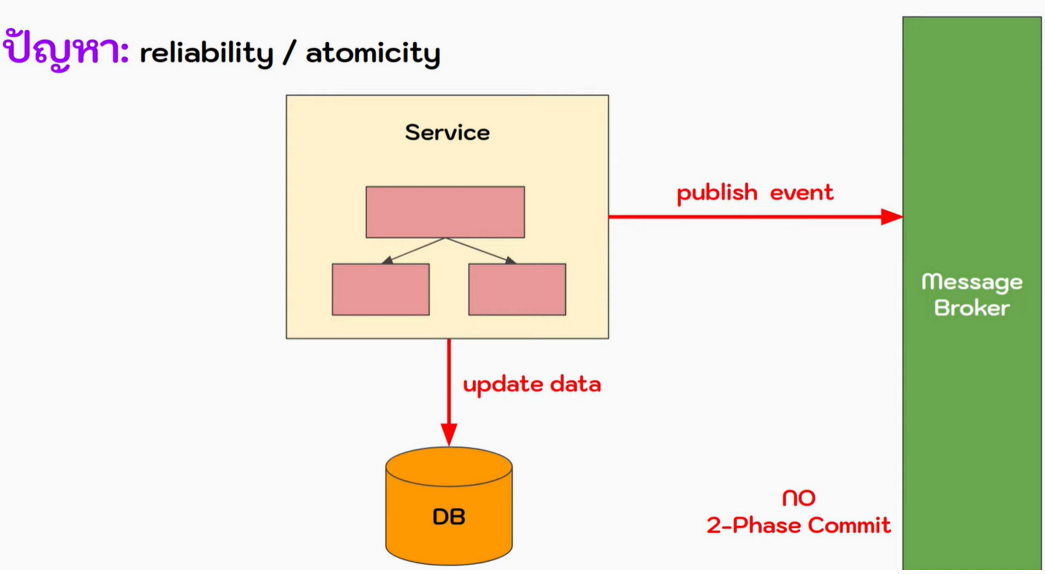
ไม่สำเร็จ

If I publish event first, what if the order unsuccessful?



But if we publish after, so like we updated the order to the database first but the event cannot be publish somehow this event will be lost forever.

Problem:
We need to update data to database along with published the event to msg broker if one fails another must also be canceled.



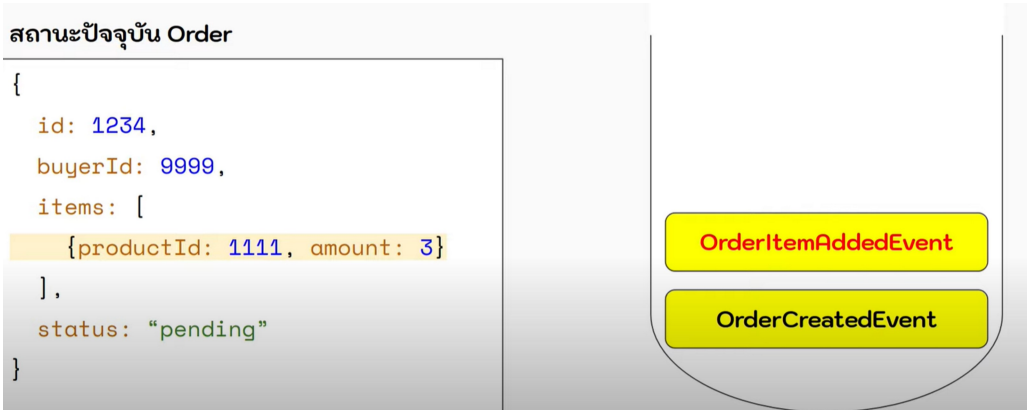
Some msg broker support 2-phase commit but it will have high latency/ slow.
So first solution is

Event Sourcing

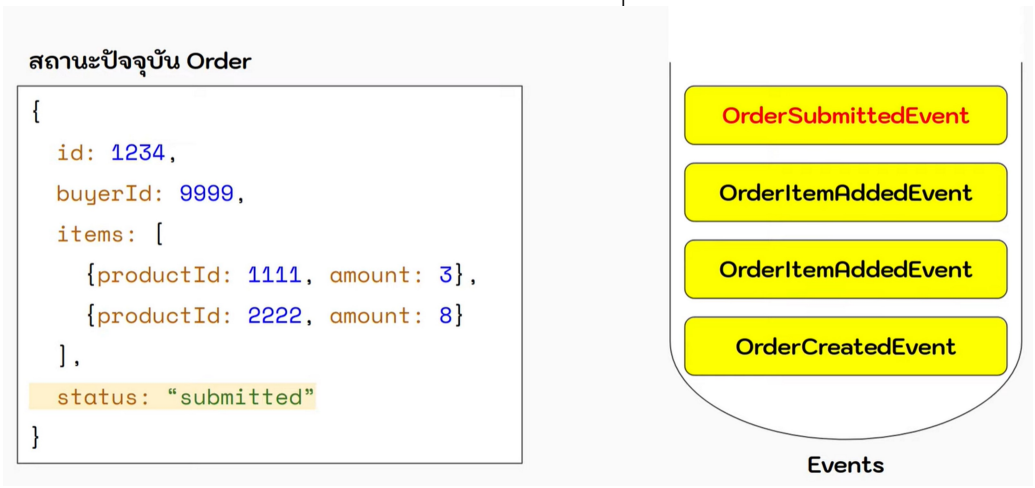
There's an idea that event that happen to an obj can be replace as status of that obj.
Like I have an empty obj, user then create an event

Status of obj order will be updated like picture above

Then user said add an item to the order

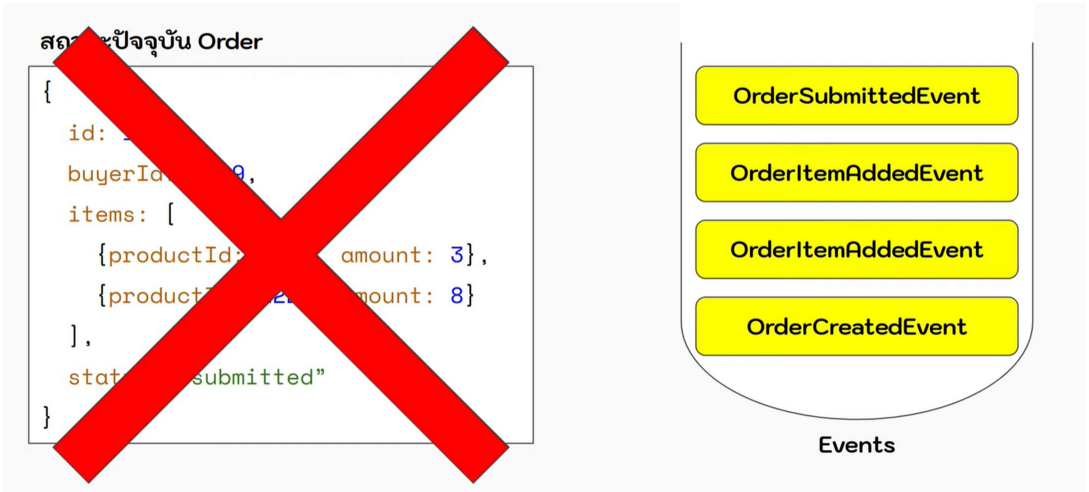


So new addItem event status of order will be updated



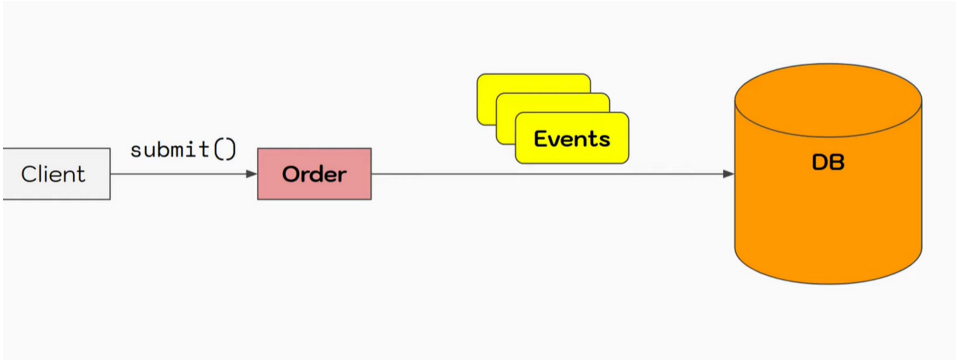
So status of order came from how we apply each event to it accordingly. If inside each event already have crucial info like order created event has buyer id. Even if we don't know the current status of order we can still compute it by reading all events.

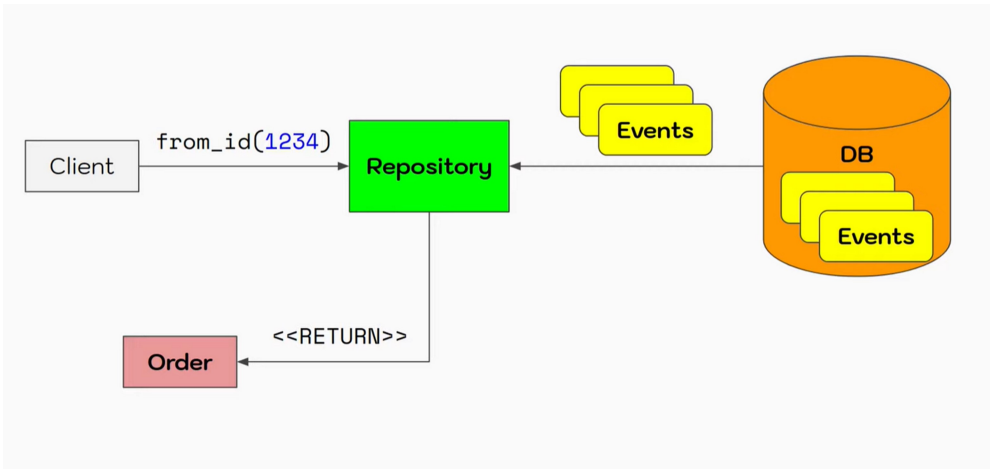
But what if we don't collect the status but the event instead



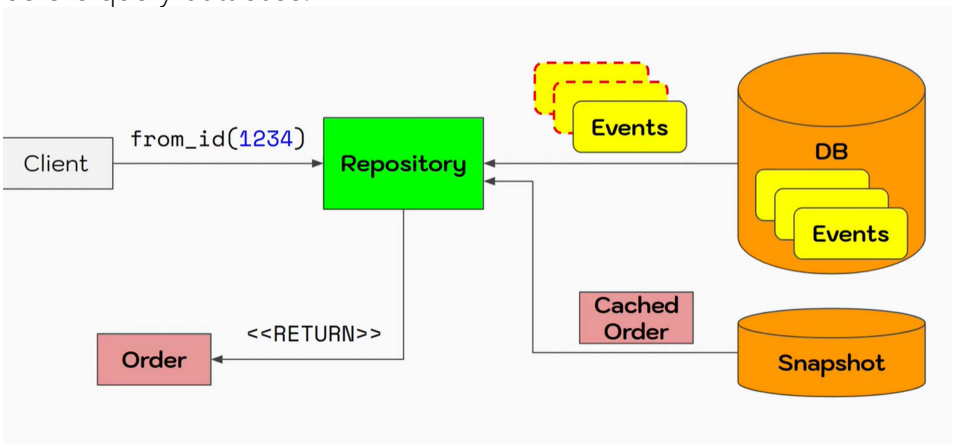
How the fuck do you do it?

Example using repository pattern.
We have a class of order, user proceed to do something, but when repository save status to the database, we save the event instead. Meaning class order will have to remember what happen to it like a list of event or something. At the end save to database.

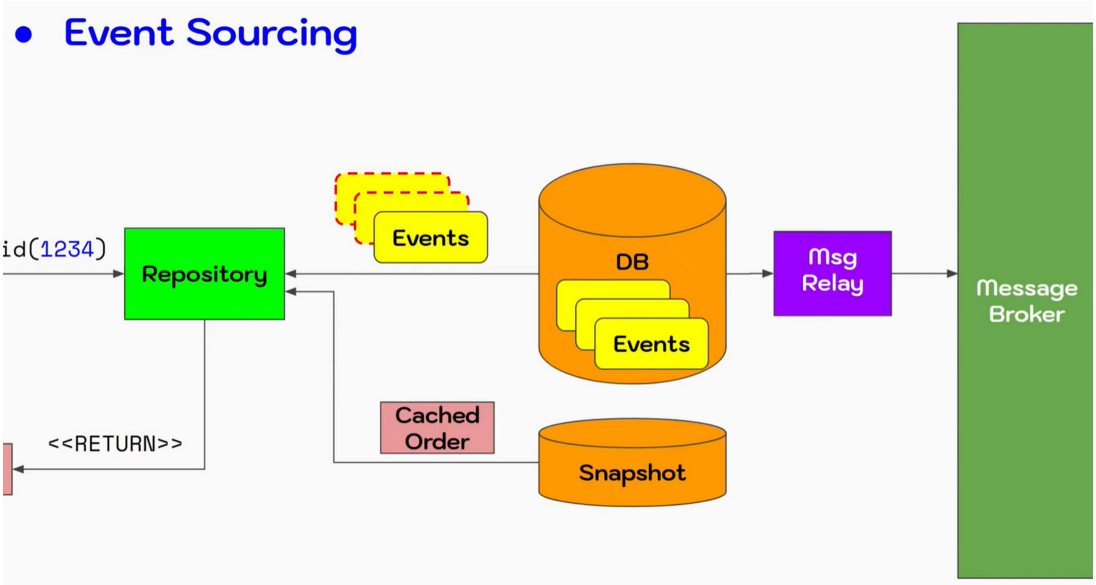




When retrieving data, repository will read all events that happen to order 1234 and compute accordingly like before, we will get the current status of this order. But reading all event might be slow so we have cached like every 5 event or something. Repo will read from here first before query database.

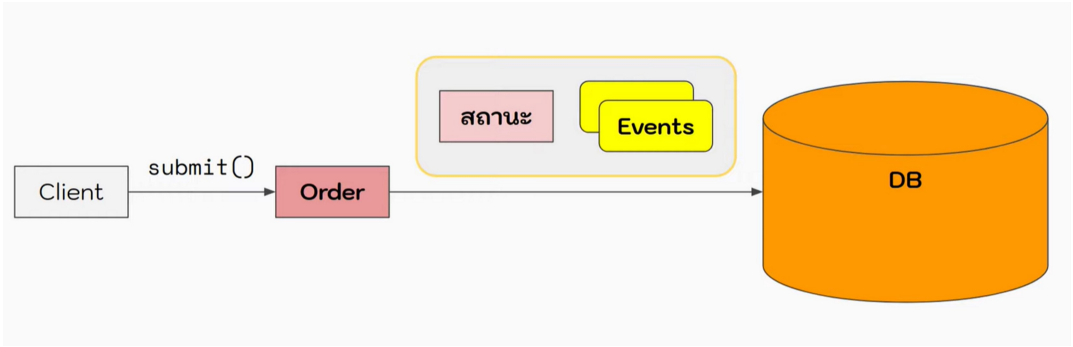


So how does this fix the problem we have about event and order?

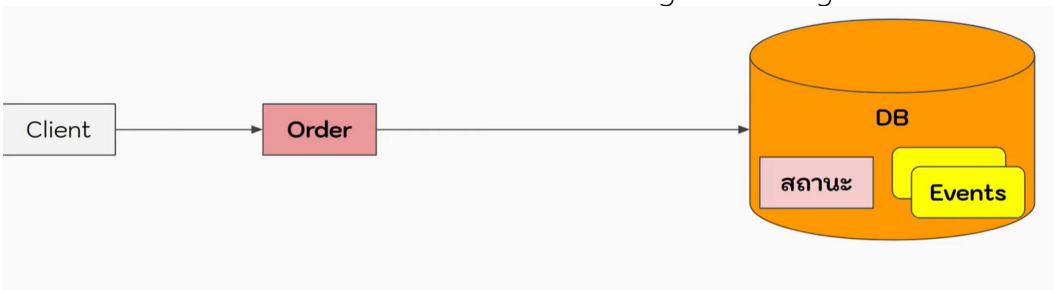


With event sourcing we don't need to collect the current status of order but instead collect the event so we can publish event from database with process or thread standby to read event from database and chose event to be publish to msg broker. We can be sure that the published event was the event that actually happen. Another pro is we can use it as debug log.
We typically use event sourcing along side CQRS pattern.

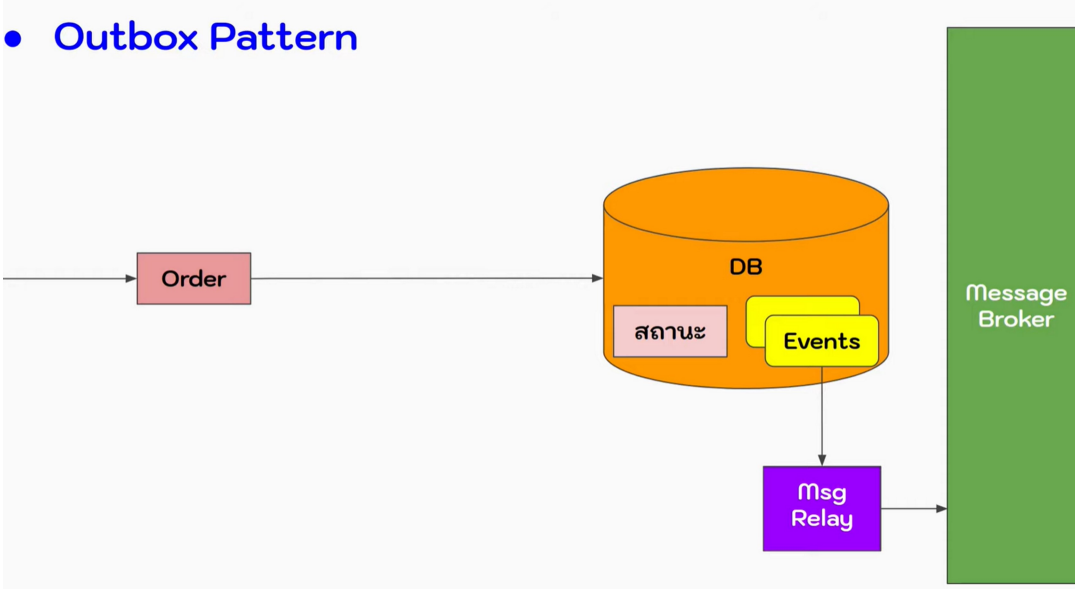
The 2nd sol is
Outbox Pattern
Similar to before we have repo pattern, but when we save the obj to the database we will save the status along with the event. So obj have to remember event and save it as one transaction.



So we can be sure that event in the database will go according to it's status

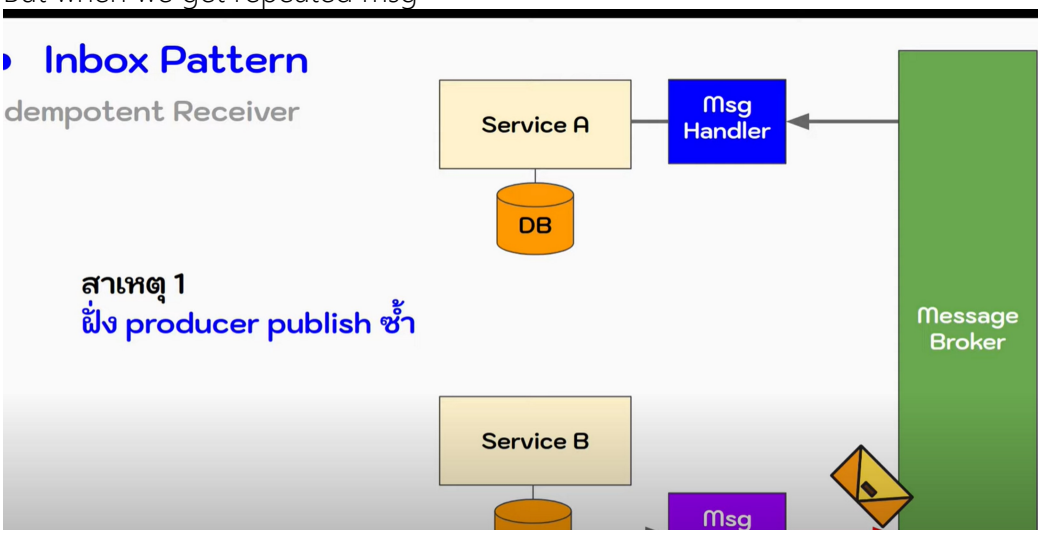


So when we publish to broker we can do it just like before

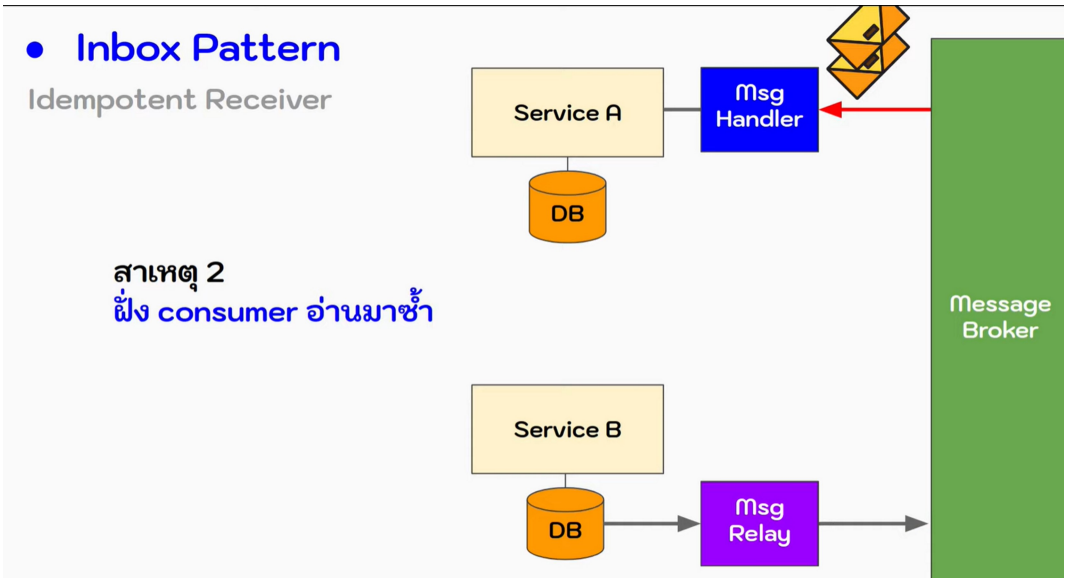


Publish with thread to broker then delete from database like event sourcing, but this time we also have status. The difference is that the event in event sourcing is event of every shits that happen in that obj, but here event is the event we want to publish. Con is database must support transaction some nonSQL cannot do it.

But when we get repeated msg



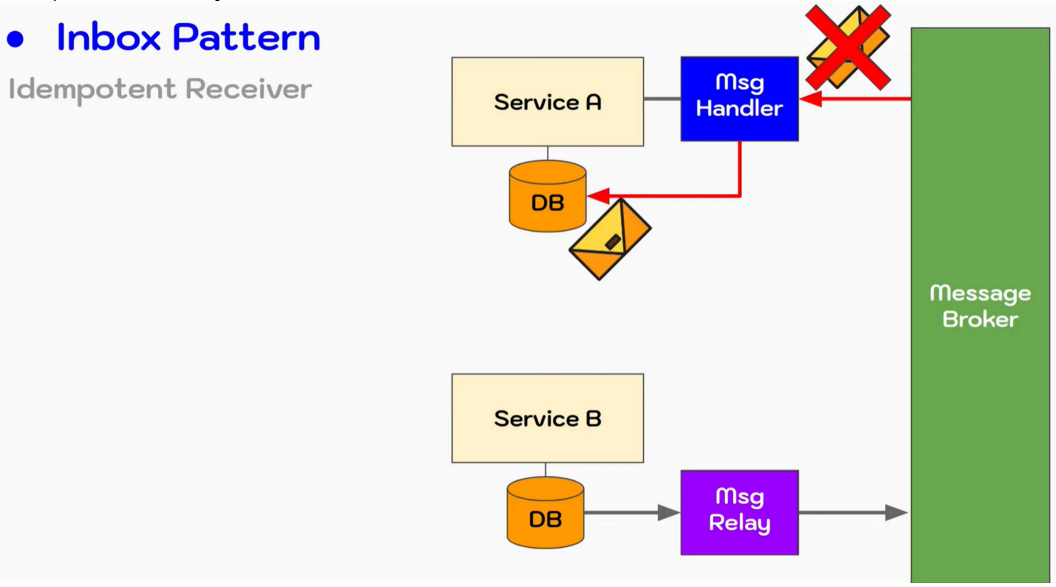
Program crash after the msg is send and is about to be deleted from database. Program revive send the msg in the database again.



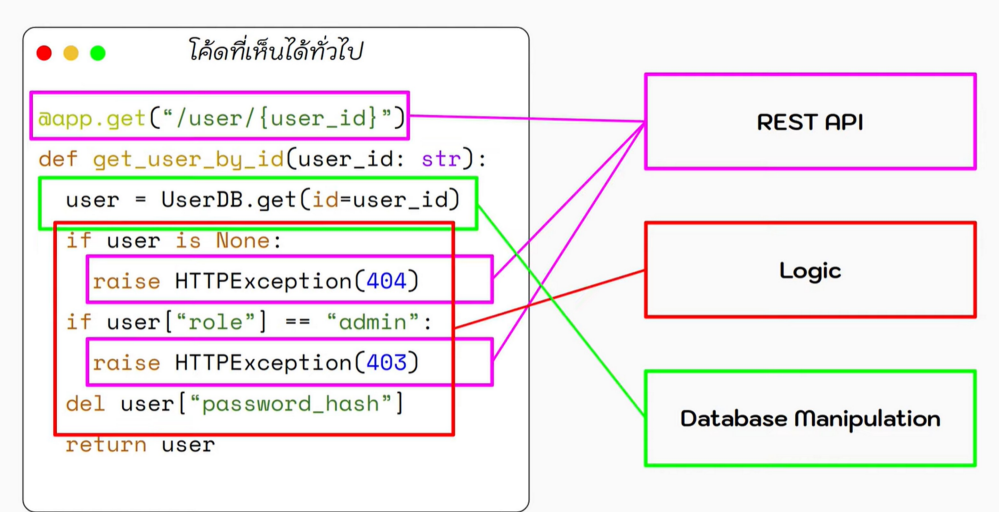
To fix issue 2 we need Idempotent receiver, receiver can receive dup msg without any consequences, one way to do it is inbox pattern

Inbox Pattern

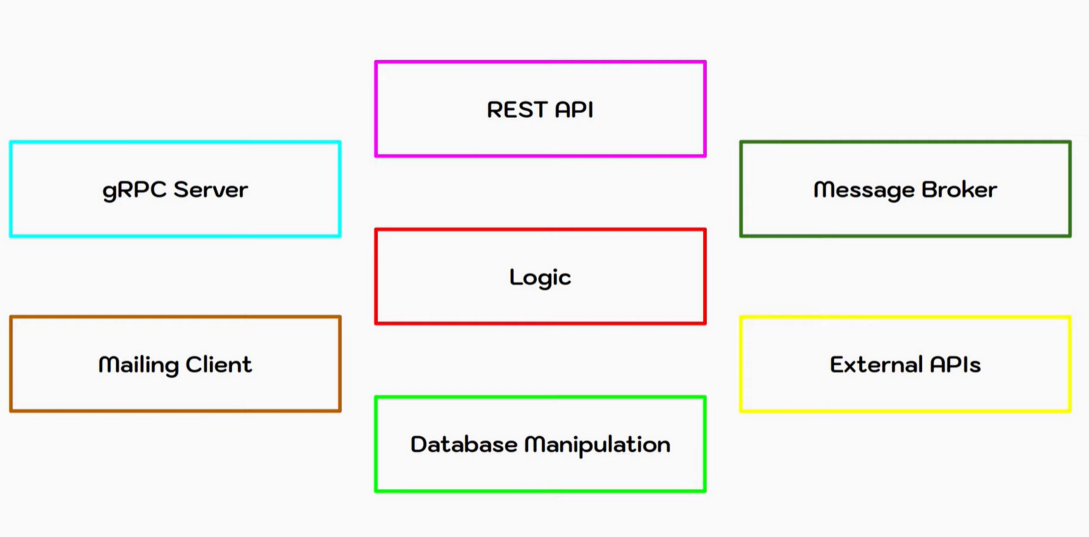
Similar to outbox, when we get msg or event to process we will save a unique id of event along with it, then we just make it check every time with this unique id of event that is this unique id already exist in the database or not.

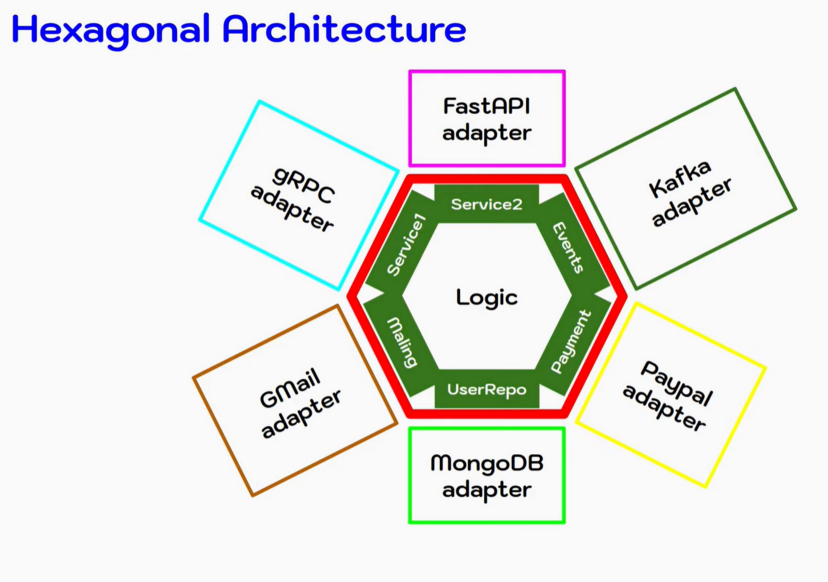


Hexagonal Architecture

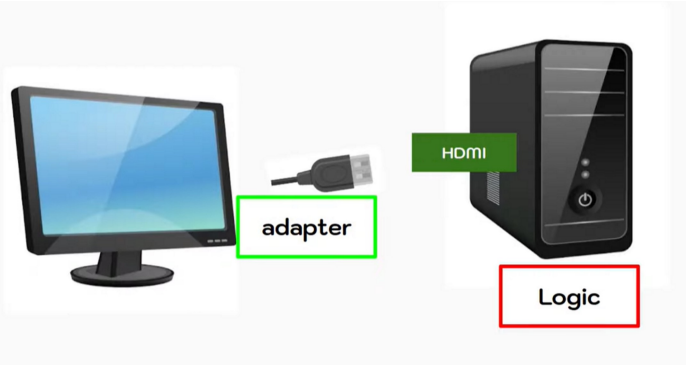


This code is a spaghetti. Hard to edit one part.
There are also many more system

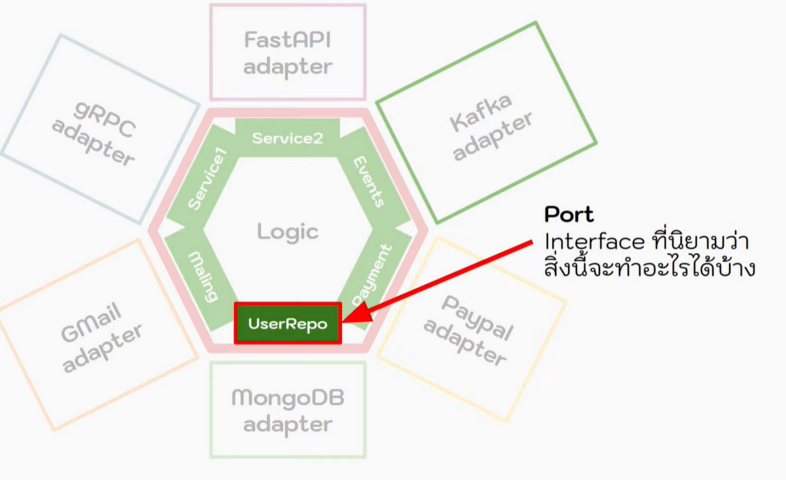




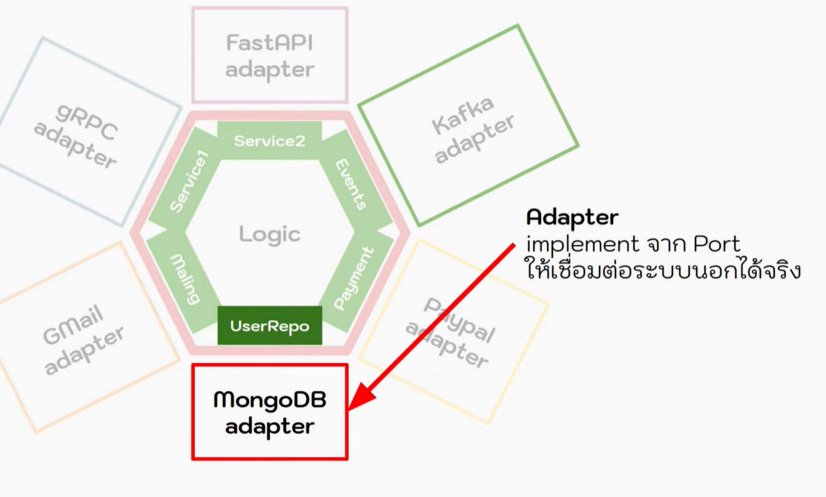
Similar to clean arch, main point is not only we separate shit we can also write logic that not depend on any external system



Imaging our system as pc and logic is CPU.

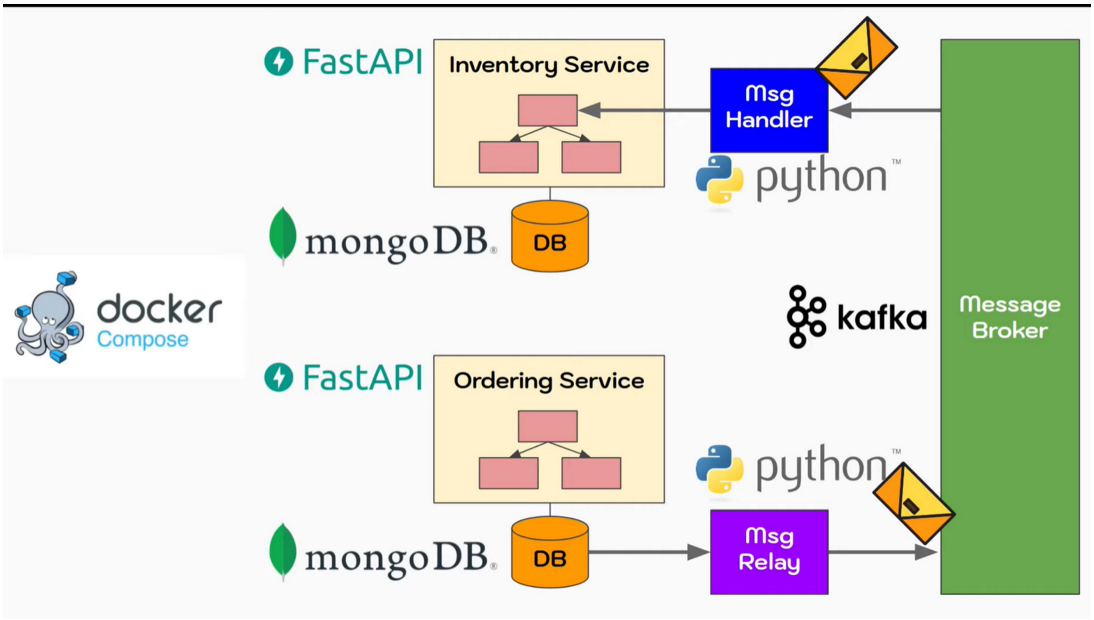


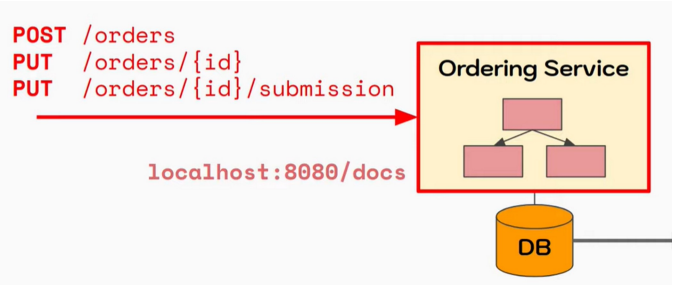
Just like that, we have port.

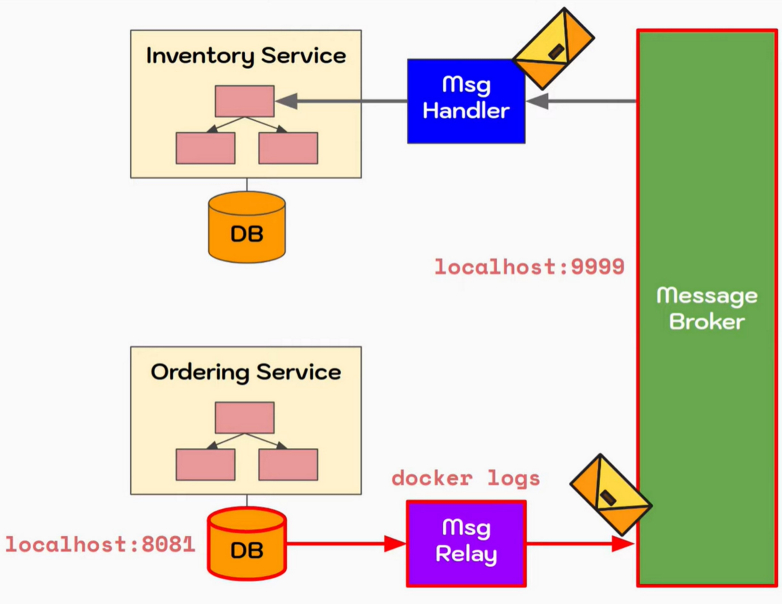


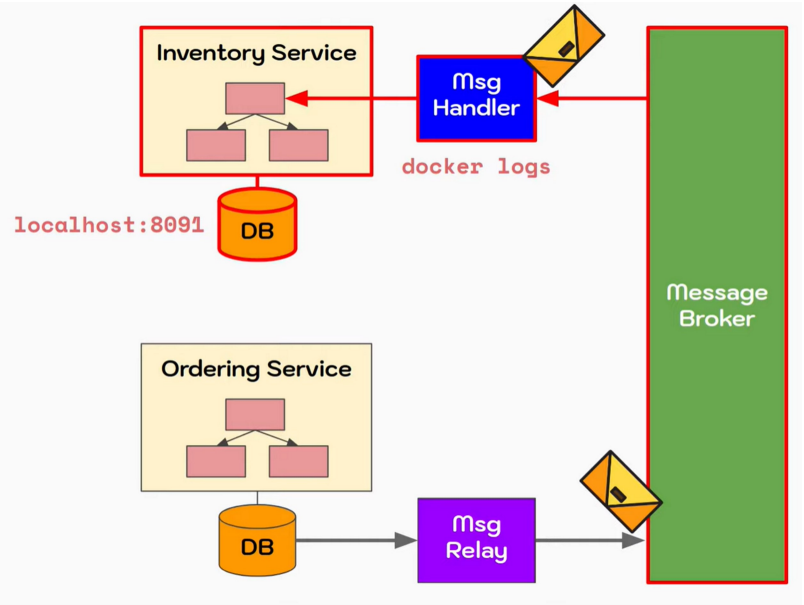
Connect shit to external shit. So logic will be separated, edit or change easily like when you want to replace mongo with mySQL we just change adapter.

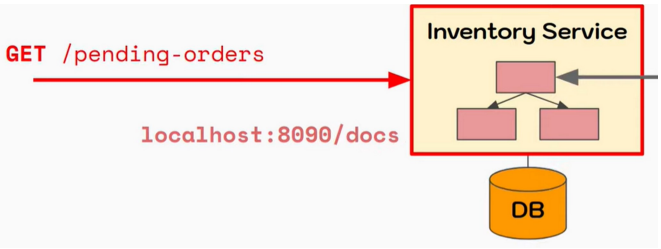
Demo

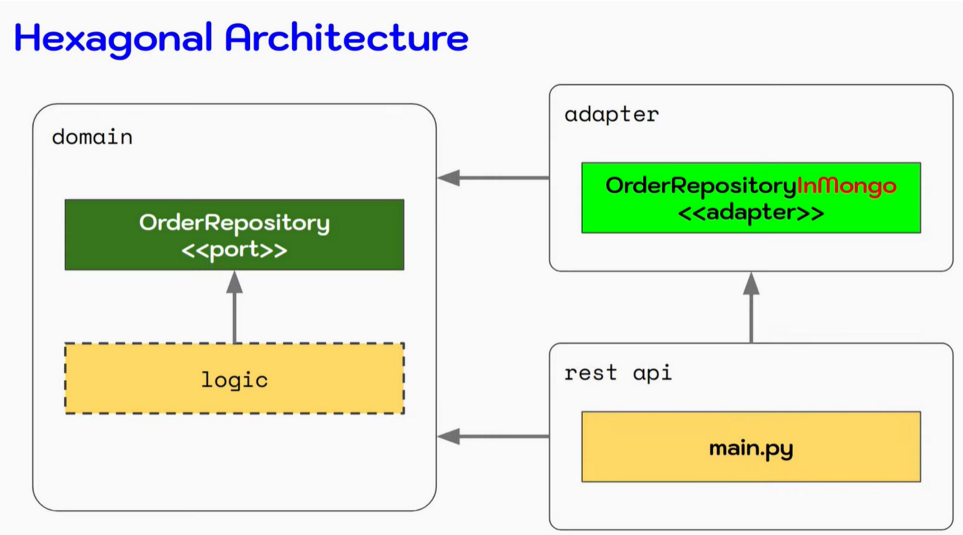

FastAPI
Inventory Service
Msg Handler
python
mongoDB.
DB
docker Compose
kafka
Message Broker
FastAPI
Ordering Service
python
mongoDB.
DB
Msg Relay

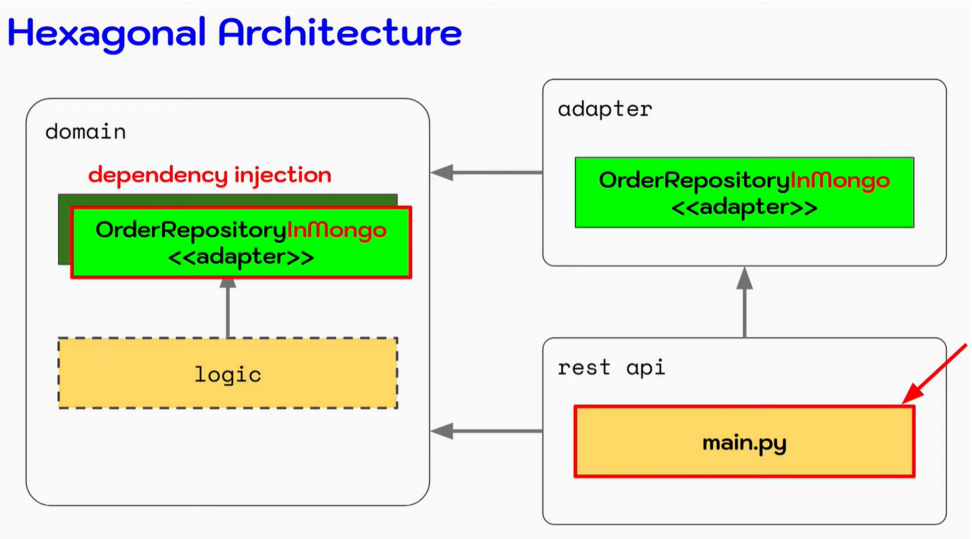

POST /orders
PUT /orders/{id}
PUT /orders/{id}/submission
Ordering Service
localhost:8080/docs
DB


Inventory Service
Msg Handler
DB
localhost:9999
Message Broker
Ordering Service
docker logs
localhost:8081
DB
Msg Relay


Inventory Service
Msg Handler
docker logs
localhost:8091
DB
Message Broker
Ordering Service
DB
Msg Relay


GET /pending-orders
Inventory Service
localhost:8090/docs
DB


Hexagonal Architecture
domain
OrderRepository
<<port>>
logic
adapter
OrderRepositoryInMongo
<<adapter>>
rest api
main.py


Hexagonal Architecture
domain
dependency injection
OrderRepositoryInMongo
<<adapter>>
logic
adapter
OrderRepositoryInMongo
<<adapter>>
rest api
main.py