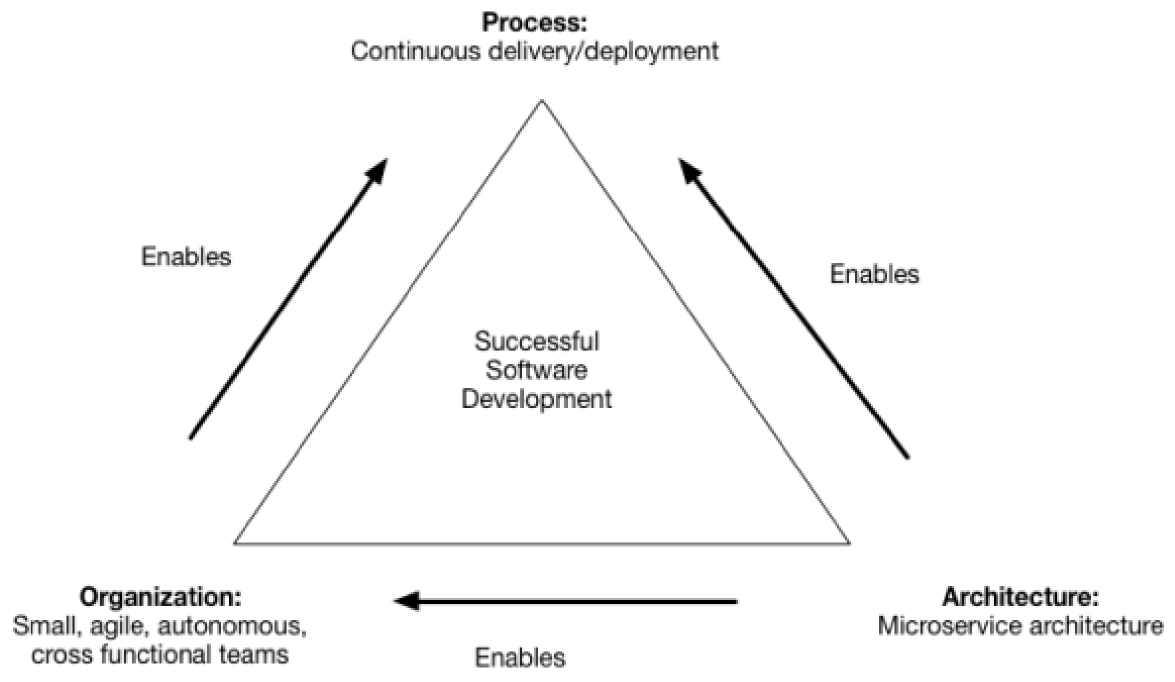
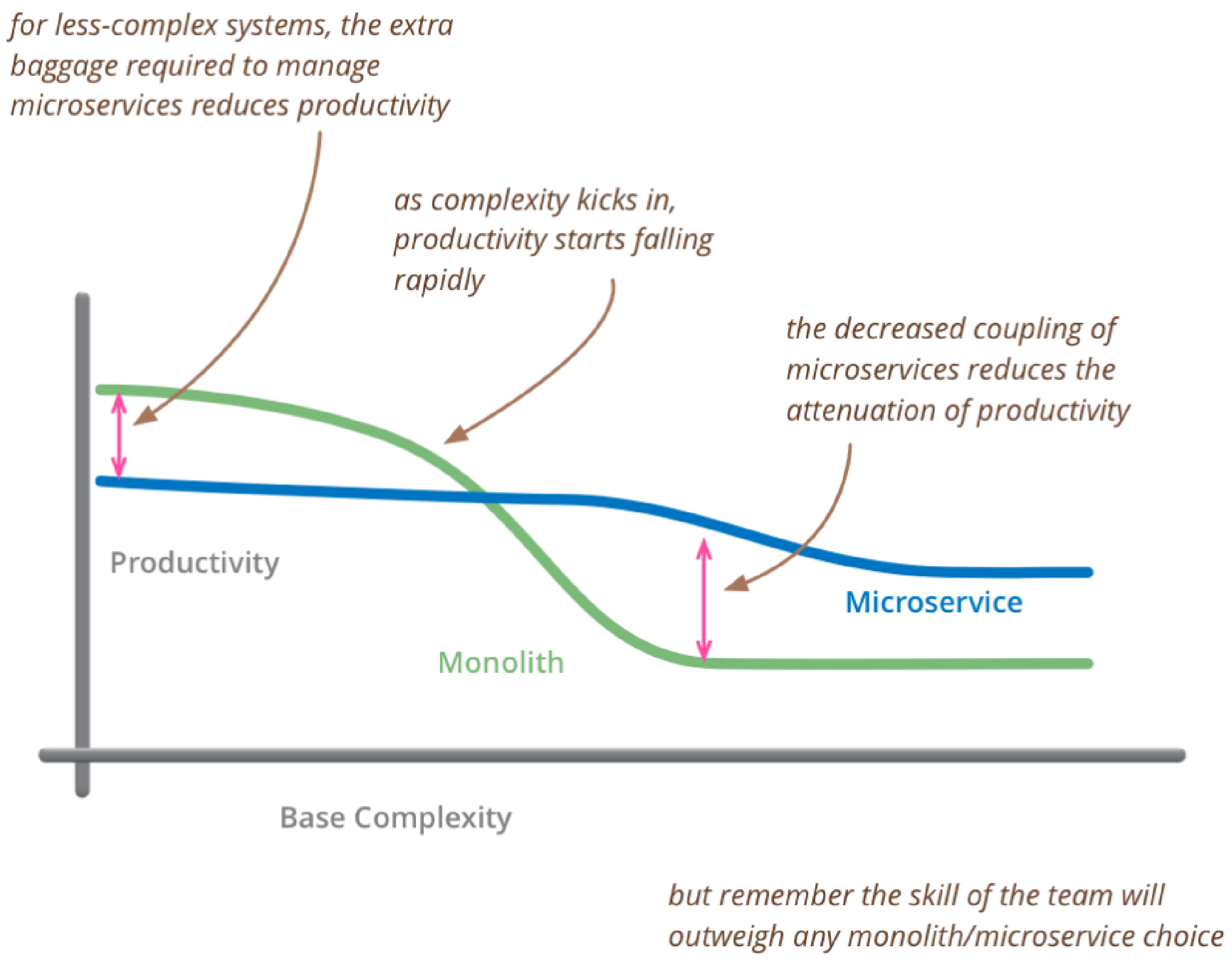
Microservices II


Process:
Continuous delivery/deployment
Enables
Enables
Successful Software Development
Organization:
Small, agile, autonomous, cross functional teams
Enables
Architecture:
Microservice architecture



https://www.cs.cmu.edu/~charlie/courses/15-214/2017-fall/slides/20171130-microservices-and-devops.pdf

Rule & Workflow

Following are the principles that need to be taken care of while developing a microservice.

- **High Cohesion**– All the business models need to be sub-divided into the smallest business part as much as possible. Each service should be focused to perform only one business task.
- **Independent**– All the services should be full stack in nature and independent of each other.
- **Business Domain Centric**– Software will modularize according to the business unit and is not tier based.
- **Automation**– Testing deployment will be automated. Try to introduce minimal human interaction.
- **Observable**– Each service will be full stack in nature and they should be independently deployable and observable like an enterprise application.

Team Management

“Two Pizza Rule” is a kind of rule that restricts the number of attendees in a microservice development team. According to this rule, number of the team members of one application should be so small such that they can be fed by two pizza. Generally, the number should not be more than 8. As microservice is full stack in nature, the team is also full stack in nature. To increase the productivity, we need to build one team of maximum 8 members with all kinds of expertise required for that service.

Task Management

> Task is an important role in software development life cycle. Developing a large scale application can be broken down into several small units of task. Let us consider we need to develop one application such as Facebook. Then, “Log in” functionality can be considered as a task of the entire build process. Progress for each of these tasks need to be monitored properly under highly skilled professionals. Agile is the well-known process structure followed in the industries to keep up with good task management.

Till now we have learned what is Microservice and what are the basic needs of it above the modern MVC architecture. In this chapter, we will learn the different elements of this architecture that are equally important for a service.

Categories of Services

By the name Microservice, we assume that it will be a service that can be consumed over HTTP protocols, however we need to know what kind of services can be build using this architecture. Following is the list of services that can be implemented using Microservice architecture.

Platform as a Service [PaaS] − In this service-oriented architecture, the platform is given as a tool which can be customized according to the business needs. PaaS plays an important role in mobile application development. The greatest example of PaaS is Google App engine, where Google provides different useful platform to build your application. PaaS originally develops to provide a built-in architecture or infrastructure to developers. It reduces the higher level programming complexity in dramatically reduced time. Following is a snapshot of Google provided PaaS.

Software as a Service [SaaS] − Software as a Service is a software licensing business, where the software is centrally hosted and licensed on a subscription basis. SaaS can be accessed mainly through the browser and it is a very common architecture pattern in many business verticals such as Human Resource Management (HRM), Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), etc. Following screenshot shows examples of different SaaS provided by Oracle.

Infrastructure as a Service [IaaS] − Infrastructure plays a good role in IT industries. Using cloud computing, some of the organizations provide virtual infrastructure as their services. IaaS is very helpful for bringing agility, cost-effectiveness, security, performance, productivity, etc. in software development. Amazon EC2 and Microsoft Azure are the biggest examples of IaaS. The following image depicts an example of AWS, where the data center is provided as IaaS.

Data as a Service [DaaS] − Information technology deals with data and some of the top industry leaders believe that data will be the new sustenance of the society. DaaS is a type of service where data is shared with business conglomerates for research and analysis. DaaS brings simplicity, agility, and security in the data access layer. Following is an example of Oracle Data cloud, which can be accessed or licensed for your own business needs.

Back End as a Service [BaaS] − BaaS is also known as MBaaS, which means mobile back-end as a service. In this type of service, backend of the application will be provided to business units for their own business ventures. All push notifications, social networking services fall under this type of services. Facebook and Twitter are examples of well-known BaaS service provider.

Security

When it comes to dealing with tons of customer data, security plays an important role. Security issue is associated with all kinds of services available in the market. Whatever the cloud you are using - private, public, hybrid, etc., security should be maintained at all levels. Entire security issue can be broadly sub-divided into the following parts −

- **Security issue faced by service providers**− This type of security issue is faced by the service providers such as Google, Amazon, etc. To ensure security protection, background check of the client is necessary especially of those who have direct access to the core part of the cloud.
- **Security issue faced by consumers**− Cloud is cost friendly, hence it is widely used across industries. Some organizations store the user details in third party data centers, and pull the data whenever required. Hence, it is mandatory to maintain security levels such that any private data of one customer should not be visible to any other users.

To prevent the above-mentioned security problems, following are some of the defensive mechanisms used by organizations.

- **Deterrent Control**− Know you potential threat to reduce cyber-attack.

- **Preventive Control**– Maintain high level authentication policy to access your cloud.
- **Detective Control**– Monitor your users and detect any potential risk.
- **Corrective Control**– Work closely with different teams and fix the issues that arise during the detective control phase.

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.
• The network is reliable
• Latency is zero
• Bandwidth is infinite
• The network is secure
• Topology doesn't change
• There is one administrator
• Transport cost is zero
• The network is homogeneous

Before applying microservices, you should have in place
• Rapid provisioning
• Dev teams should be able to automatically provision new infrastructure
• Basic monitoring
• Essential to detect problems in the complex system landscape
• Rapid application deployment
• Service deployments must be controlled and traceable
• Rollbacks of deployments must be easy

• Microservice architectures enable independent evolution of services – but how is this done without breaking existing clients?
• There are two answers
• Version service APIs on incompatible API changes
• Using JSON and REST limits versioning needs of service APIs
• Versioning is key
• Service interfaces are like programmer APIs – you need to know which version you program against
• As service provider, you need to keep old versions of your interface operational while delivering new versions • But first, let's recap compatibility

There are two types of compatibility
• Forward Compatibility
    • Upgrading the service in the future will not break existing clients

    • Requires some agreements on future design features, and the design of new versions to respect old interfaces

• Backward Compatibility
    • Newly created service is compatible with old clients
    • Requires the design of new versions to respect old interfaces The hard type of compatibility is forward compatibility!

Forward compatibility through REST and JSON
REST and JSON have a set of inherent agreements that benefit forward compatibility
• JSON: only validate for what you really need, and ignore unknown object fields (i.e. newly introduced ones)
• REST: HATEOAS links introduce server-controlled indirection between operations and their URIs

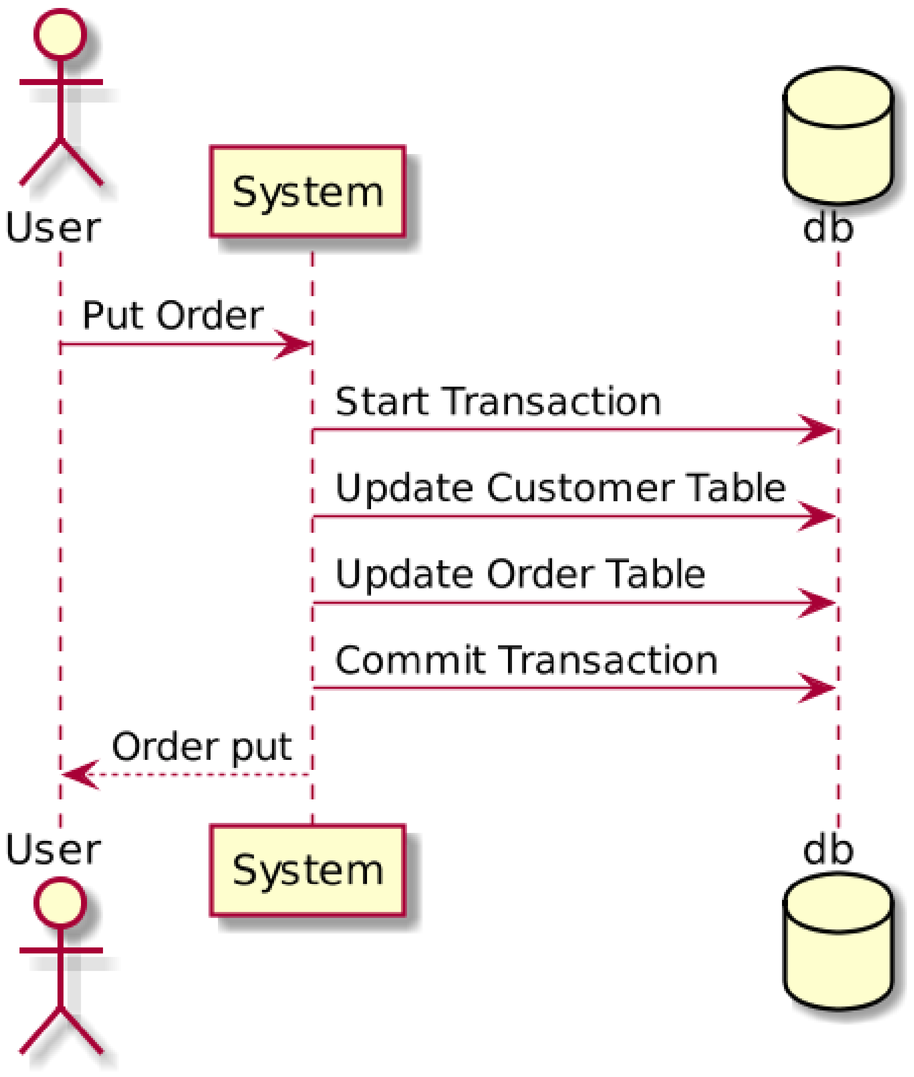
Compatibility and Versioning
Compatibility can't be always guaranteed, therefore versioning schemes (major.minor.point) are introduced
• Major version change: breaking API change
• Minor version change: compatible API change Note that versioning a service imposes work on the service provider
• Services need to exist in their old versions as long as they are used by clients
• The service provider has to deal with the mapping from old API to new API as long as old clients exist

Further Challenges
• Testing the whole system
    • A single microservice isn't the whole system.
    • A clear picture of upstream and downstream services is needed for integration testing
• Transactions
    • Instead of distributed transactions, compensations are used (as in SOA)
• Authentication
    • Is often offloaded to reverse proxies making use auf authentication (micro)services
• Request logging
    • Pass along request tokens
    • Add them to the log
    • Perform log aggregation

What is a distributed transaction?

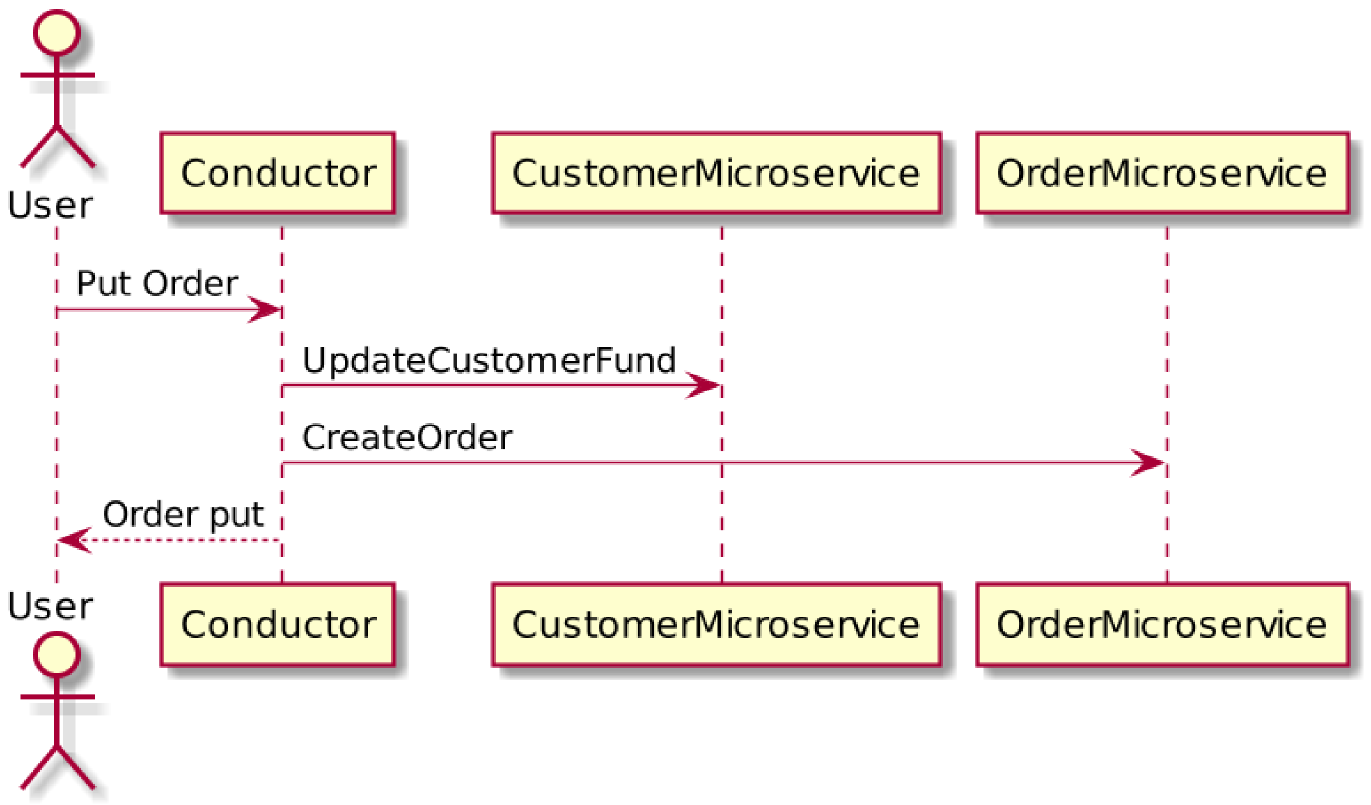
When a microservice architecture decomposes a monolithic system into self-encapsulated services, it can break transactions. This means a **local transaction** in the monolithic system is now **distributed** into multiple services that will be called in a sequence.

Here is a customer order example with a monolithic system using a local transaction:



https://medium.com/codex/compensating-transaction-in-microservices-15b1f88a7c29
In the customer order example above, if a user sends a **Put Order** action to a monolithic system, the system will create a local database transaction that works over multiple database tables. If any step fails, the transaction can **roll back**. This is known as ACID.

When we decompose this system, we created both the `CustomerMicroservice` and the `OrderMicroservice`, which have separate databases. Here is a customer order example with microservices:



In monolithic application we have database system to ensure Acidity but the question is how to ensure atomic transaction in case of microservices application ?

In this model, a typical business operation consists of a series of separate steps. While these steps are being performed, the overall view of the system state might be inconsistent, but when the operation has completed and all of the steps have been executed the system should become consistent again. So in microservices the idea is to achieve eventual consistency .

A challenge in the eventual consistency model is how to handle a step that has failed. In this case it might be necessary to undo all of the work completed by the previous steps in the operation. However, the data can't simply be rolled back because other concurrent instances of the application might have changed it. Even in cases where the data hasn't been changed by a concurrent instance, undoing a step might not simply be a matter of restoring the original state.

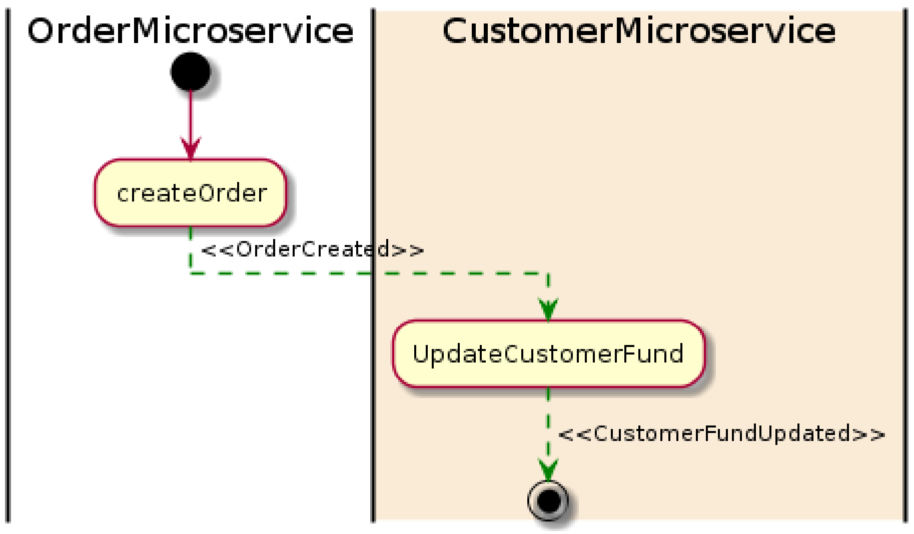
Solution :

The solution is to implement a compensating transaction. The steps in a compensating transaction must undo the effects of the steps in the original operation. A compensating transaction might not be able to simply replace the current state with the state the system was in at the start of the operation because this approach could overwrite changes made by other concurrent instances of an application. Instead, it must be an intelligent process that takes into account any work done by concurrent instances.

Compensating transaction can be achieved by using SAGA design pattern .

The Saga pattern is another widely used pattern for distributed transactions. It is different from 2pc, which is synchronous. The Saga pattern is asynchronous and reactive. In a Saga pattern, the distributed transaction is fulfilled by asynchronous local transactions on all related microservices.

Here is a diagram of the Saga pattern for the customer order example:



In the example above, the `OrderMicroservice` receives a request to place an order. It first starts a local transaction to create an order and then emits an `OrderCreated` event. The `CustomerMicroservice` listens for this event and updates a customer fund once the event is received. If a deduction is successfully made from a fund, a `CustomerFundUpdated` event will then be emitted, which in this example means the end of the transaction.

If any microservice fails to complete its local transaction, the other microservices will run compensation transactions to rollback the changes. Here is a diagram of the Saga pattern for a compensation transaction:

In the above example, the `UpdateCustomerFund` failed for some reason and it then emitted a `CustomerFundUpdateFailed` event. The `OrderMicroservice` listens for the event and start its compensation transaction to revert the order that was created.

- A client that initiates the saga, which an asynchronous flow, using a synchronous request (e.g. HTTP `POST /orders`) needs to be able to determine its outcome. There are several options, each with different trade-offs:
  - The service sends back a response once the saga completes, e.g. once it receives an `OrderApproved` or `OrderRejected` event.
  - The service sends back a response (e.g. containing the `orderID`) after initiating the saga and the client periodically polls (e.g. `GET /orders/{orderID}`) to determine the outcome

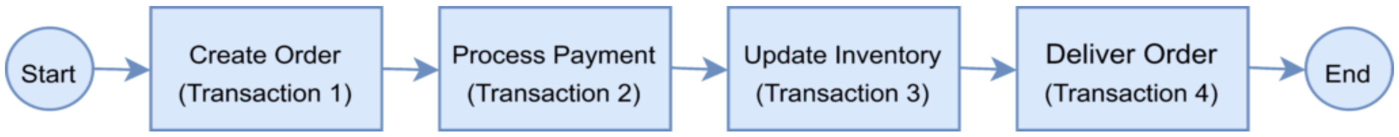
- The service sends back a response (e.g. containing the `orderID`) after initiating the saga, and then sends an event (e.g. websocket, web hook, etc) to the client once the saga completes.

3. Distributed Transaction

To demonstrate the use of distributed transactions, we'll take an example of an e-commerce application that processes online orders and is implemented with microservice architecture.

There is a microservice to create the orders, one that processes the payment, another that updates the inventory and the last one that delivers the order.

Each of these microservices performs a local transaction to implement the individual functionalities:


Start
Create Order (Transaction 1)
Process Payment (Transaction 2)
Update Inventory (Transaction 3)
Deliver Order (Transaction 4)
End

This is an example of a distributed transaction as the transaction boundary crosses multiple services and databases.

To ensure a successful order processing service, all four microservices must complete the individual local transactions. If any of the microservices fail to complete its local transaction, all the completed preceding transactions should roll back to ensure data integrity.

https://www.baeldung.com/cs/saga-pattern-microservices

4. Challenges of Distributed Transaction

In the previous section, we provided a real-life example of a distributed transaction. Distributed transactions in a microservice architecture pose two key challenges.

The first challenge is maintaining ACID. To ensure the correctness of a transaction, it must be Atomic, Consistent, Isolated and Durable (ACID). The *atomicity*ensures that all or none of the steps of a transaction should complete. *Consistency*takes data from one valid state to another valid state. *Isolation*guarantees that concurrent transactions should produce the same result that sequentially transactions would have produced. Lastly, *durability*means that committed transactions remain committed irrespective of any type of system failure. **In a distributed transaction scenario, as the transaction spans several services, ensuring ACID always remains key.**

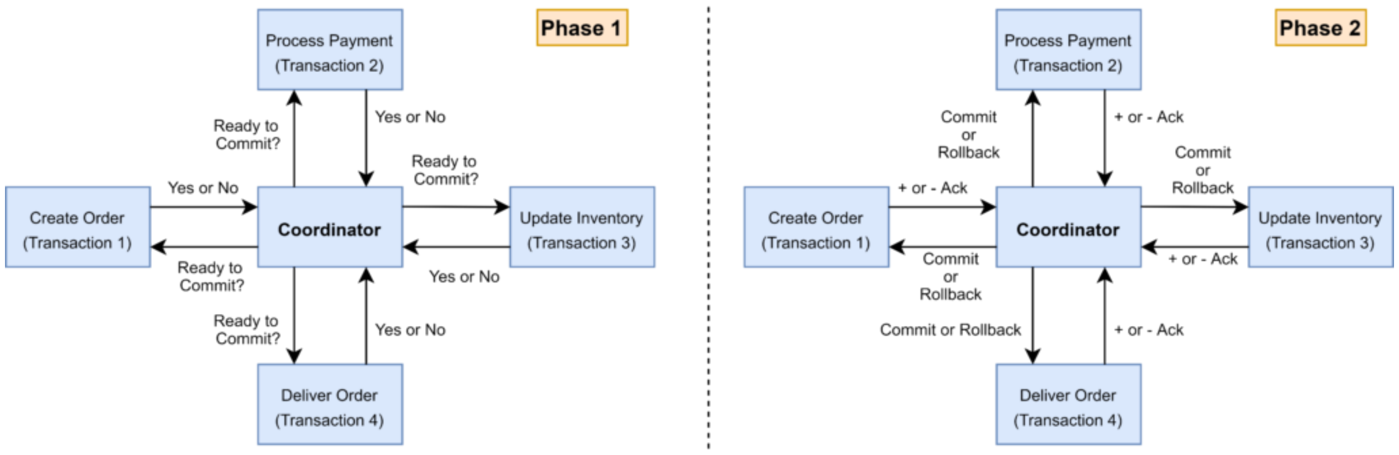
The second challenge is managing the transaction isolation level. It specifies the amount of data that is visible in a transaction when the other services access the same data simultaneously. In other words, if one object in one of the microservices is persisted in the database while another request reads the data, should the service return the old or new data?

5. Understanding Two-Phase Commit

The Two-Phase Commit protocol (2PC) is **a widely used pattern to implement distributed transactions.** We can use this pattern in a microservice architecture to implement distributed transactions.

In a two-phase commit protocol, there is a coordinator component that is responsible for controlling the transaction and contains the logic to manage the transaction.

The other component is the participating nodes (e.g., the microservices) that run their local transactions:


Phase 1
Process Payment (Transaction 2)
Ready to Commit?
Yes or No
Create Order (Transaction 1)
Coordinator
Update Inventory (Transaction 3)
Deliver Order (Transaction 4)
Phase 2
Commit or Rollback
+ or - Ack
Commit or Rollback

As the name indicates, the two-phase commit protocol runs a distributed transaction in two phases:

1. **Prepare Phase** – The coordinator asks the participating nodes whether they are ready to commit the transaction. The participants returned with a *yes* or *no*.
2. **Commit Phase** – If all the participating nodes respond affirmatively in phase 1, the coordinator asks all of them to commit. If at least one node returns negative, the coordinator asks all participants to roll back their local transactions.

6. Problems With 2PC

Although 2PC is useful to implement a distributed transaction, it has the following shortcomings:

- The **onus of the transaction is on the coordinator node**, and it can become the single point of failure.
- All other services need to wait until the slowest service finishes its confirmation. So, the overall performance of the transaction is bound by the slowest service.

- The two-phase commit protocol is **slow by design due to the chattiness and dependency on the coordinator.** So, it can lead to scalability and performance issues in a microservice-based architecture involving multiple services.
- Two-phase commit protocol is **not supported in NoSQL databases.** Therefore, in a microservice architecture where one or more services use NoSQL databases, we can't apply a two-phase commit.

Pattern: Shared database

Resulting context

The benefits of this pattern are:

- A developer uses familiar and straightforward ACID transactions to enforce data consistency
- A single database is simpler to operate

The drawbacks of this pattern are:

- Development time coupling - a developer working on, for example, the `OrderService` will need to coordinate schema changes with the developers of other services that access the same tables. This coupling and additional coordination will slow down development.
- Runtime coupling - because all services access the same database they can potentially interfere with one another. For example, if long running `CustomerService` transaction holds a lock on the `ORDER` table then the `OrderService` will be blocked.
- Single database might not satisfy the data storage and access requirements of all services.

Resulting context

Using a database per service has the following benefits:

- Helps ensure that the services are loosely coupled. Changes to one service's database does not impact any other services.
- Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j.

Using a database per service has the following drawbacks:

- Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the CAP theorem. Moreover, many modern (NoSQL) databases don't support them.
- Implementing queries that join data that is now in multiple databases is challenging.
- Complexity of managing multiple SQL and NoSQL databases