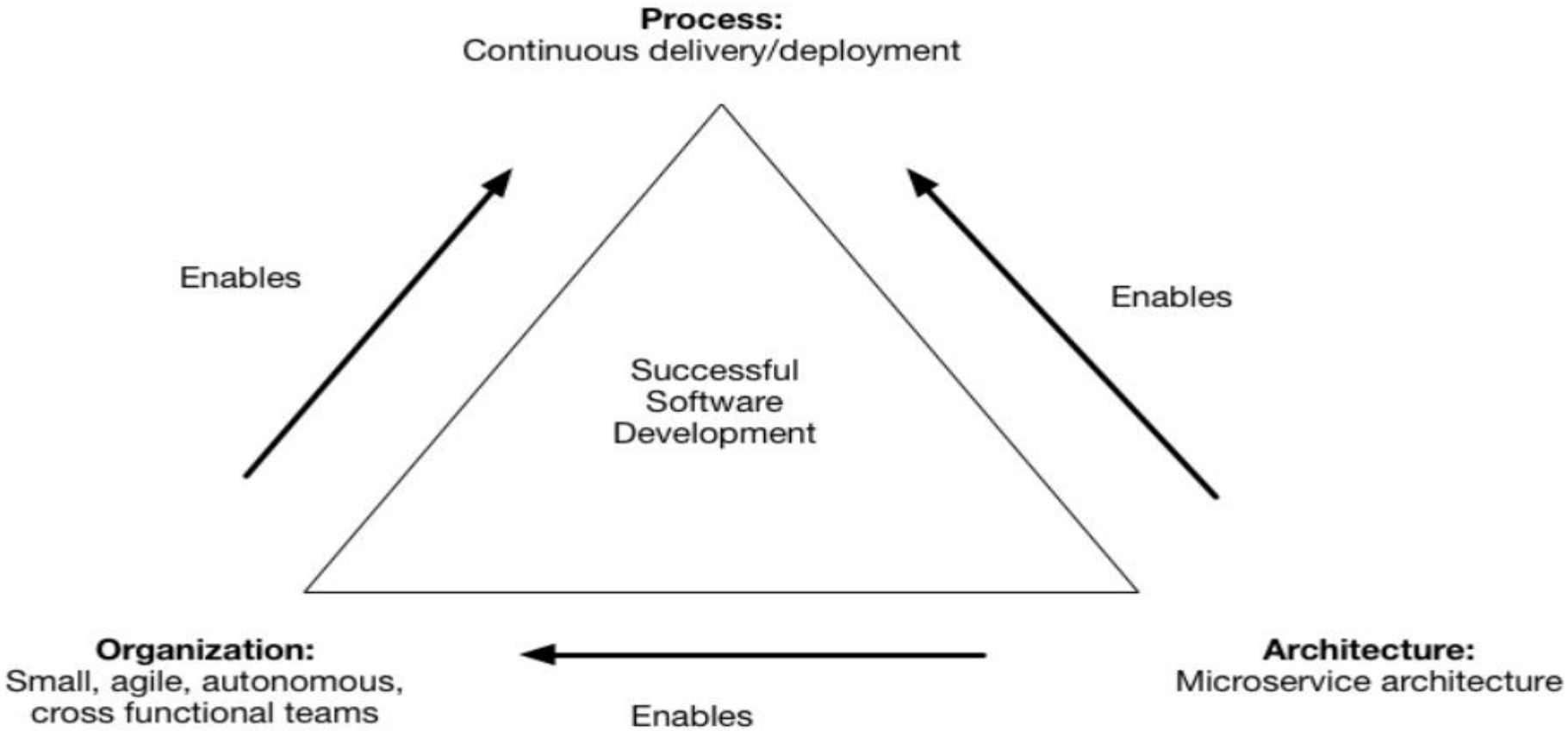
Microservice Design Steps

By
Apparao G

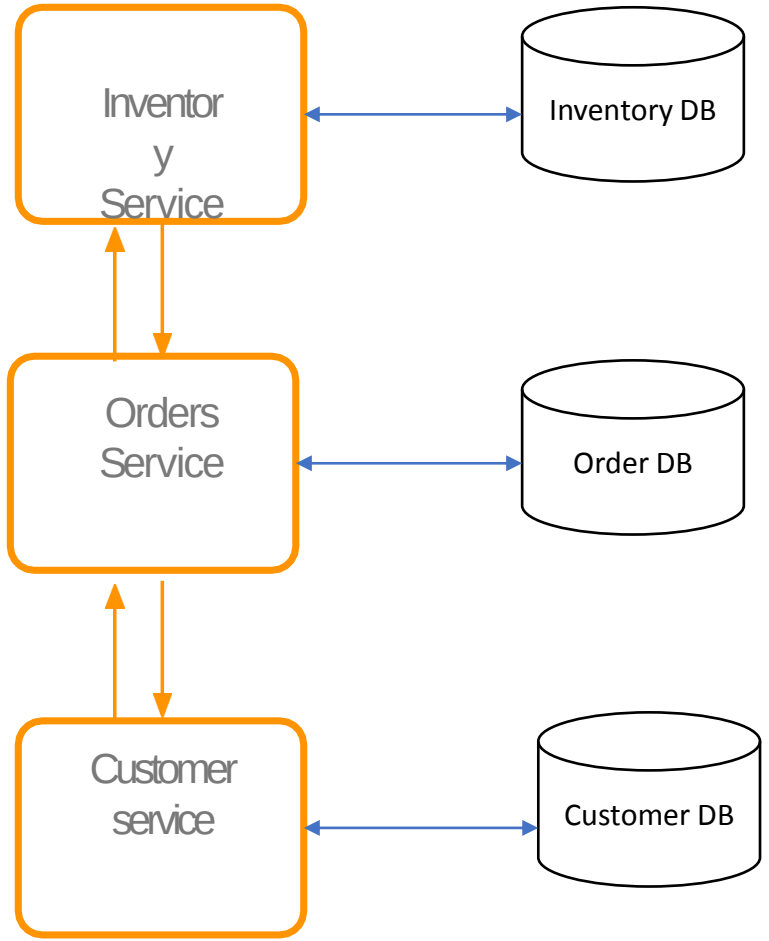
How to decompose an application into services?

1. Decompose by business capability
2. Decompose by subdomain



Data Management in microservices:

1. Database per service : The service's database is effectively part of the implementation of that service. It cannot be accessed directly by other services.



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.

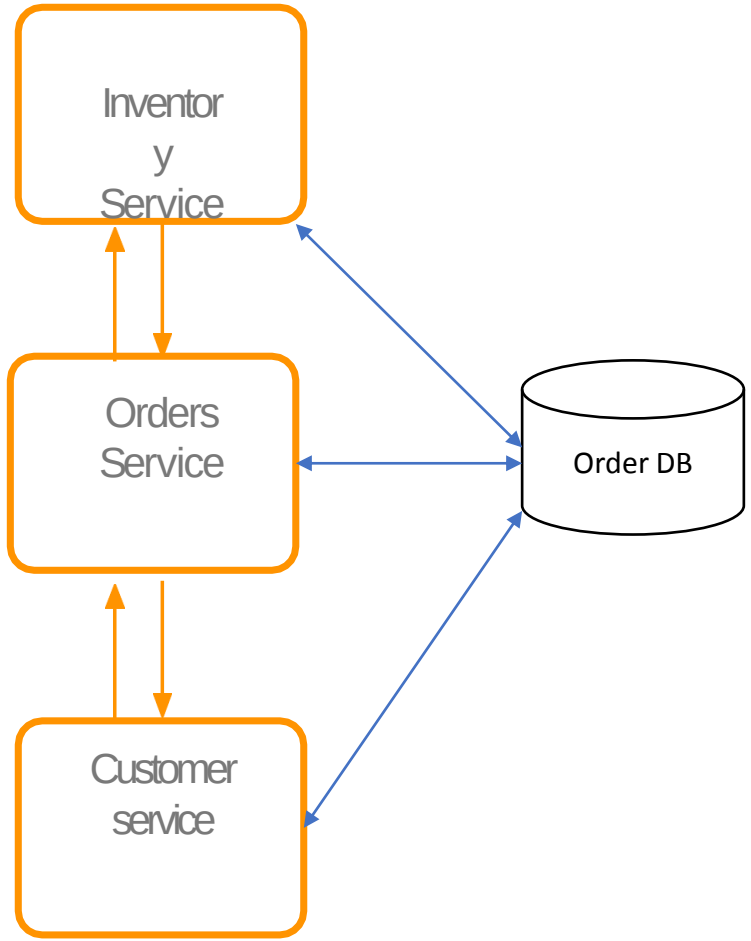
For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j.

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.

2. Shared database: Use single database that is shared by multiple services. Each service freely accesses data owned by other services using local ACID transactions.

Order Service can use the following ACID transaction to create new order :




```
BEGIN TRANSACTION
 // check inventory
 SELECT QUANTITY
 FROM PRODUCT WHERE PRODUCT_ID = ?

 // Check customer credit limit
 SELECT CREDIT_LIMIT
 FROM CUSTOMERS WHERE CUSTOMER_ID = ?

 // Create order and order items
 INSERT INTO ORDER
(ID,NAME,TOTAL_AMOUNT,CUSTOMER_ID)
                values(....)
 INSERT INTO ORDER_ITEM(ID,NAME,PRICE,
 PRODUCT_ID)  values(....)

 COMMIT TRANSACTION
```

Benefits:

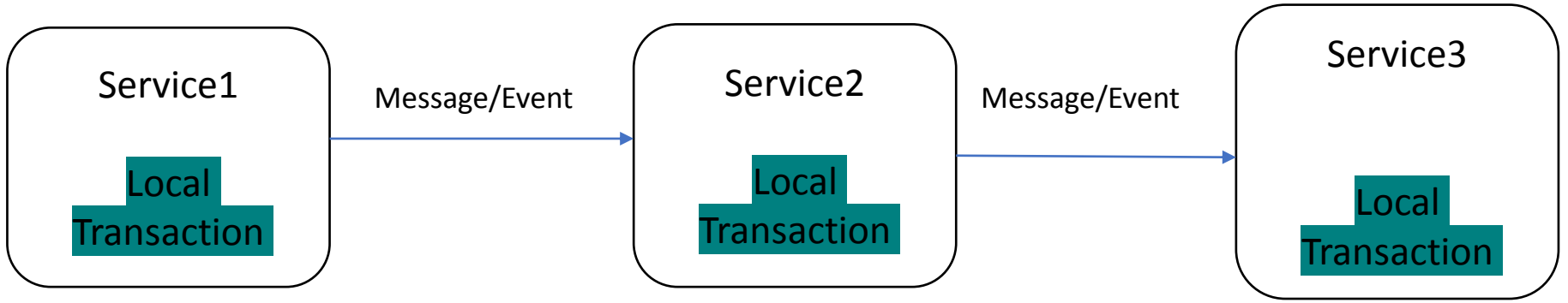
- A developer uses familiar and straightforward ACID transactions to enforce data consistency.
- A single database is simpler to operate.

Drawbacks:

- 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.

Transaction Management using Saga Pattern:

A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



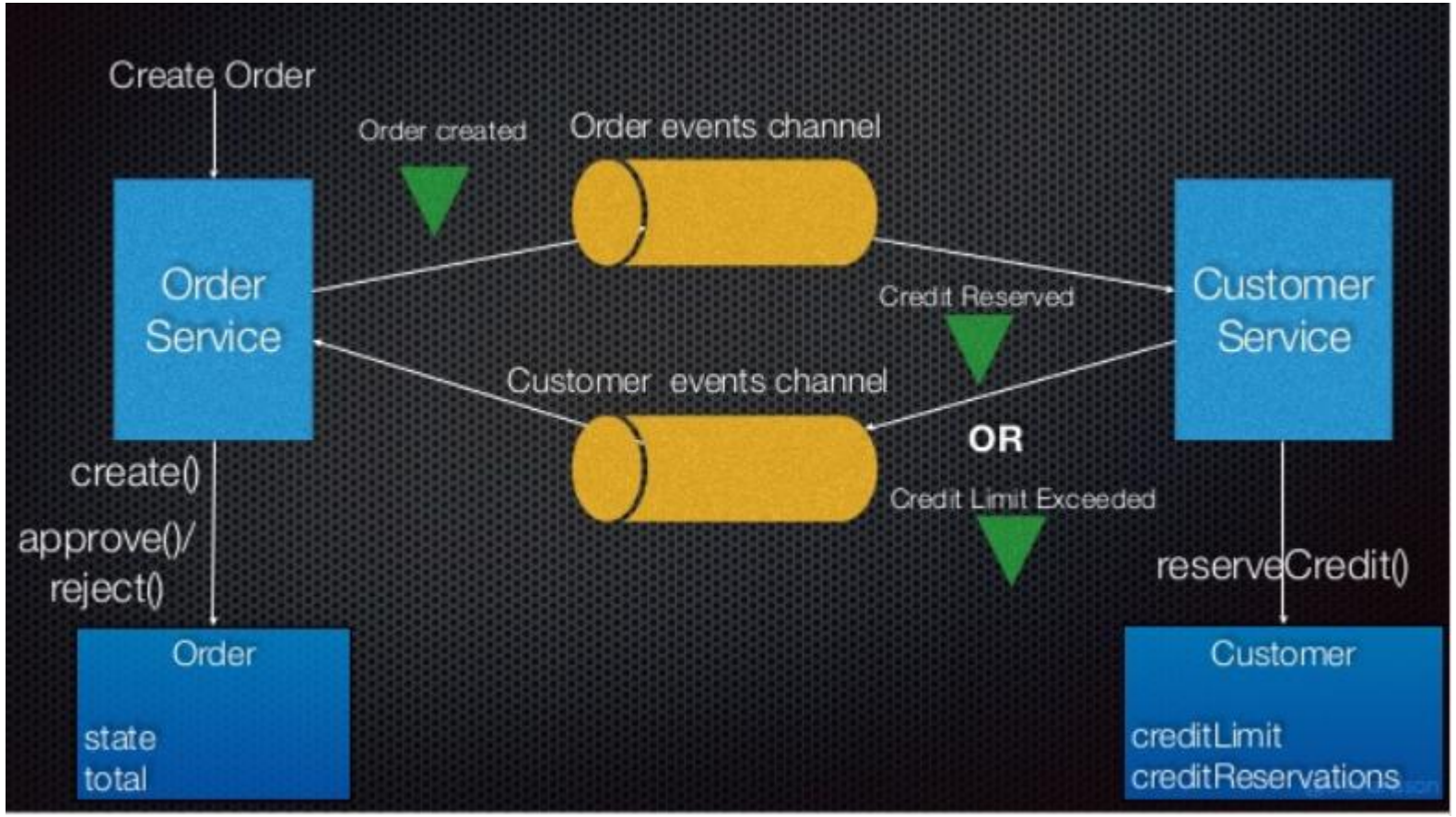
There are two ways of coordination sagas:

Choreography - each local transaction publishes domain events that trigger local transactions in other services

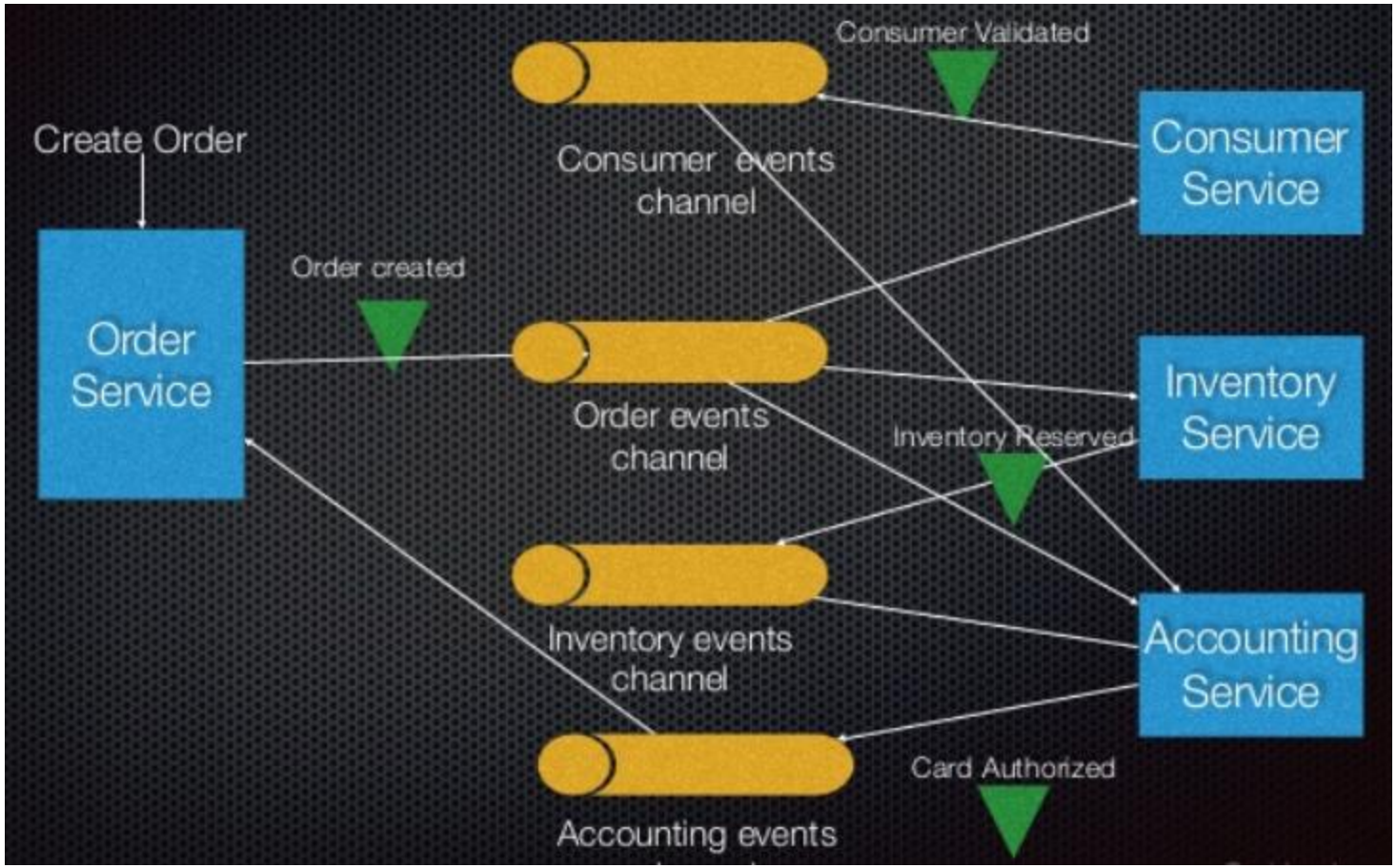
Orchestration - an orchestrator (object) tells the participants what local transactions to execute.

Choreography based coordination using events:

An e-commerce application that uses this approach would create an order using a choreography-based saga that consists of the following steps:



More complex Choreography example:



The Order Service receives the POST /orders request and creates an Order in a PENDING state

It then emits an Order Created event

The Customer Service’s event handler attempts to reserve credit

It then emits an event indicating the outcome

The OrderService’s event handler either approves or rejects the Order

Benefits:

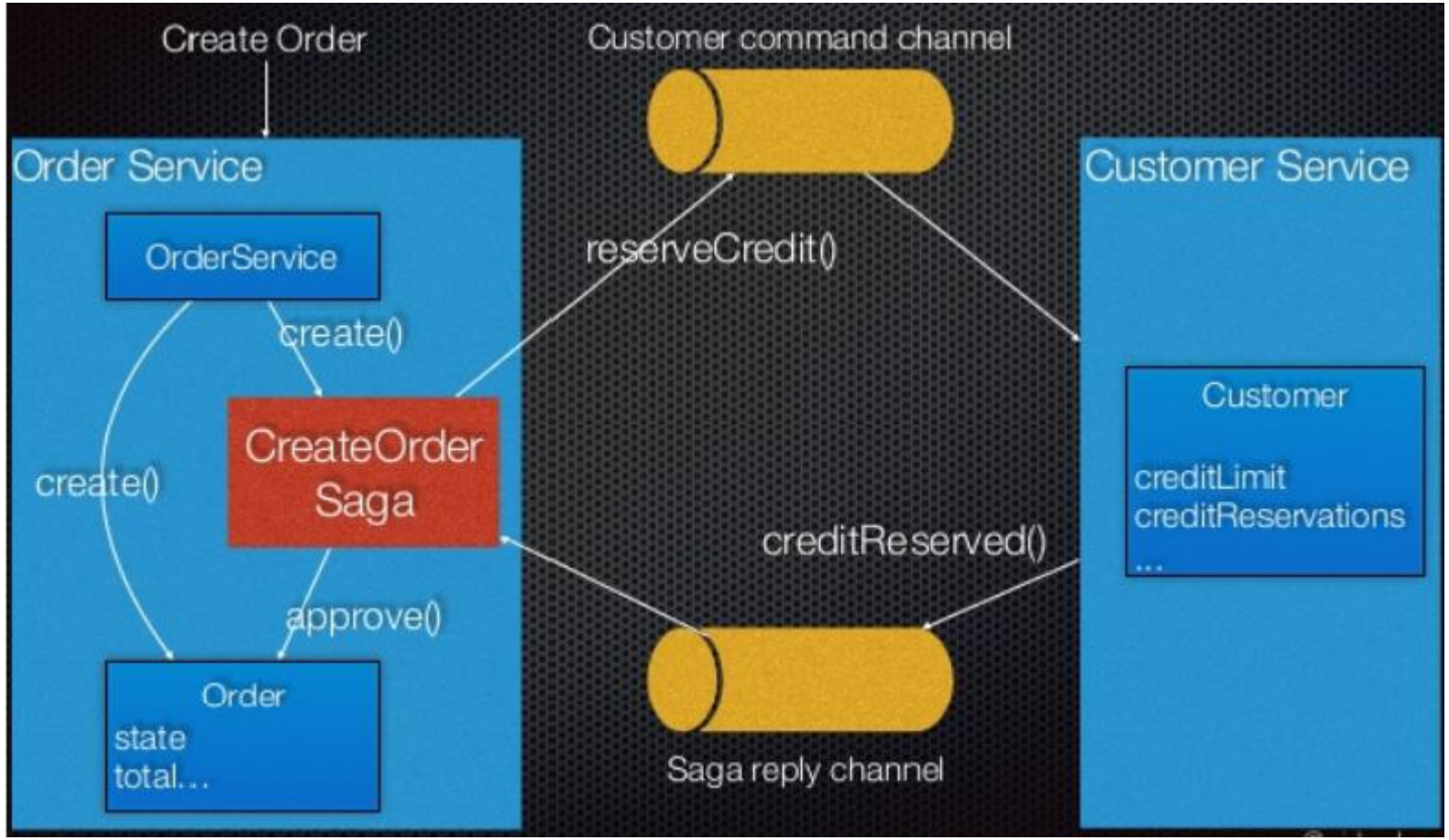
- Simple, especially when using event sourcing.
- Participants are loosely coupled.

Drawbacks:

- Cyclic dependencies -services listen to each other's events.
- Overloads domain objects.
- Events-Indirect way to make something happen.

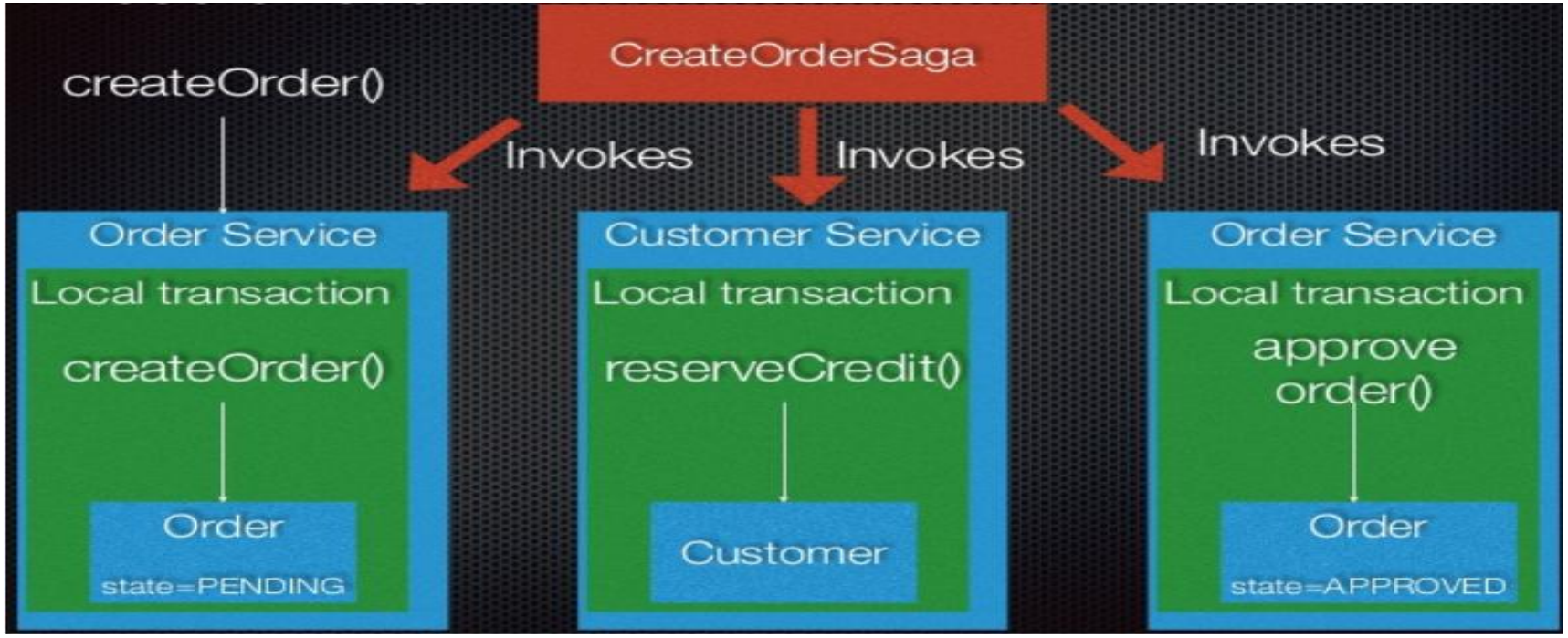
Orchestration based saga coordination:

A saga (Orchestrator) is a persistent object that tracks the state of the saga and invokes the participants.



Saga orchestrator behavior:

On create:	On replay:
• Invokes a saga participant • Persists state in database • Wait for a reply	• Load state from database • Determine which saga participant to invoke next • Invokes saga participants • Updates its state • Persists updated state • Wait for a replay



Benefits:

- It enables an application to maintain data consistency across multiple services without using distributed transactions
- Centralized coordination logic is easier to understand
- Reduces cyclic dependencies.

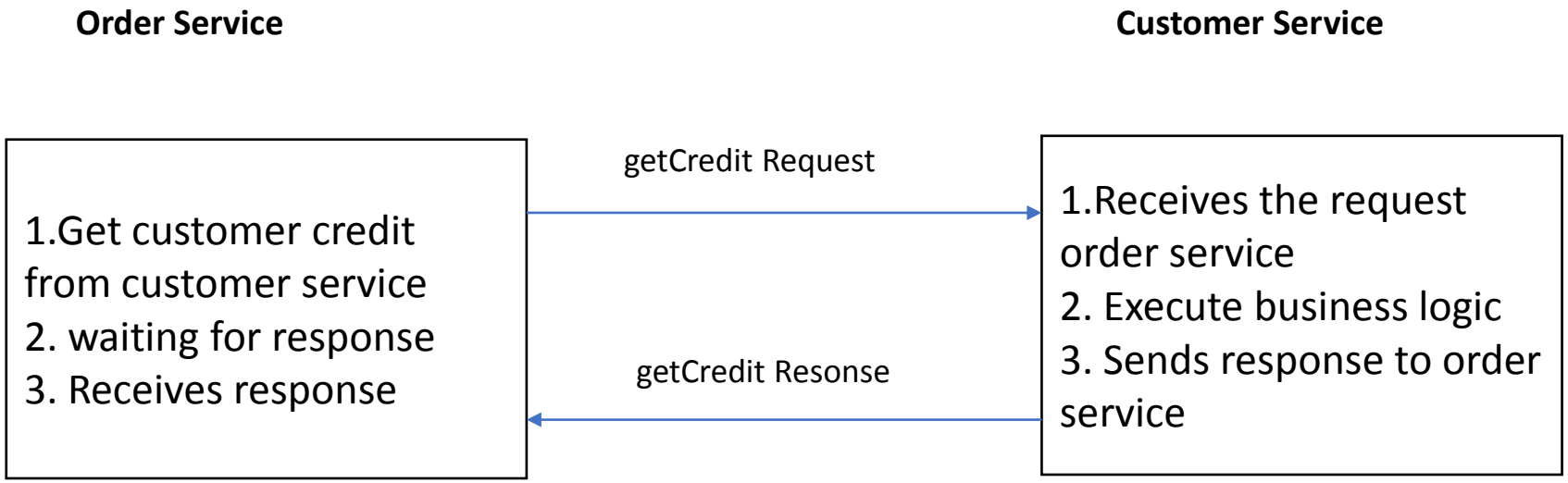
Drawback:

The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.

Microservice communication:

1.Synchronous communication:

The client can make a REST call to interact with other services. The client sends a request to the server and waits for a response from the service (Mostly JSON). For example, Spring Cloud Netflix provides the most common pattern for synchronous REST communication such as Hystrix.



In the above diagram, Order Service calls Customer Service and waits for a response returned by Customer Service. Order Service can then process customer Service's response in the same transaction that triggered the communication.

There are different protocols, such as REST, gRPC, and Apache Thrift, that can be used to interact with services synchronously.

Most synchronous communications are one-to-one. In synchronous one-to-one communication, you can also use multiple instances of a service to scale the service.

We can achieve scaling using load-balancing mechanism on the client side.

Load balancer:

- Load balancer has all information about the calling service instances.
- It checks the health of the each node (instance) , if node is up then it will send request to that node, otherwise it will not send any request to that node.
- There are several load-balancing algorithm available and those are:

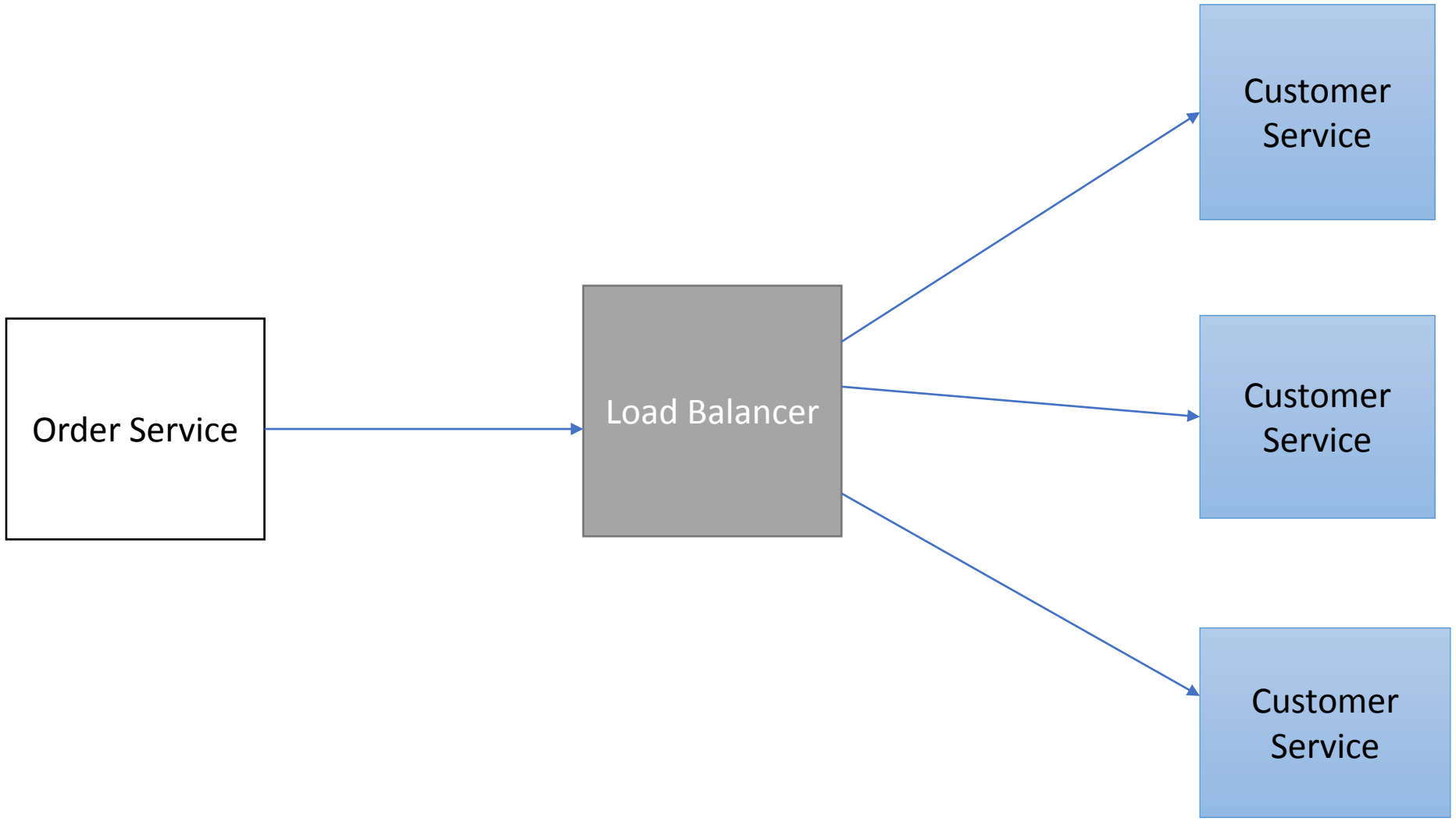
Round-Robin: This is the simplest method that routes requests across all the instances sequentially.

Least Connections: This is a method in which the request goes to the instance that has the fewest number of connections at the time.

Weighted Round-Robin: This is an algorithm that assigns a weight to each instance and forwards the connection according to this weight.

IP Hash: This is a method that generates a unique hash key from the source IP address and determines which instance receives the request.

Load Balancer:



Synchronous communication has the following benefits:

- Simple and familiar
- Request/reply is easy
- Simpler system since there in no intermediate broker

Synchronous communication has the following drawbacks:

- Usually only supports request/reply and no other interaction patterns such as notifications, request/async response, publish/subscribe, publish/async response
- Reduced availability since the client and the service must be available for the duration of the interaction

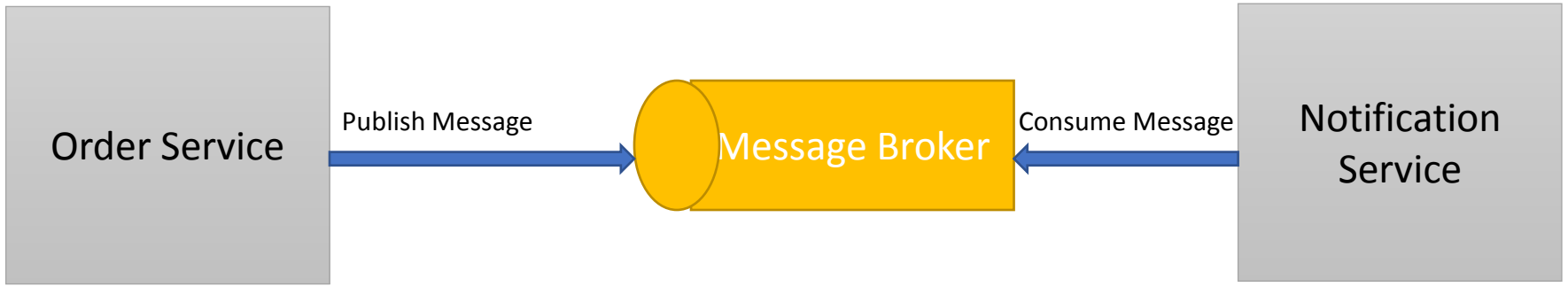
Synchronous communication sample code:

```
@Component
public class InventoryServiceUtil {

		@Autowired
		public RestTemplate restTemplate;

		@Value("${inventory.service.url}")
		public String inventoryUrl;

		@HystrixCommand(fallbackMethod = "productDetails")
		public ProductDetails getProductDetails(ProductItemDetails productItemDetails) {
				StringBuilder url = new StringBuilder(inventoryUrl).append(productItemDetails.getProductId());
				ProductDetails productDetails = restTemplate.getForObject(url.toString(), ProductDetails.class);
				return productDetails;
		}

		public ProductDetails productDetails() {
				System.out.println("Invoking fallback method...");
				ProductDetails productDetails = new ProductDetails();
				return productDetails;
		}
}
```

Asynchronous communication:

In this communication style, the client service doesn't wait for the response coming from another service. So, the client doesn't block a thread while it is waiting for a response from the server. Such type of communications is possible by using lightweight messaging brokers.

The message producer service doesn't wait for a response. It just generates a message and sends message to the broker, it waits for the only acknowledgement from the message broker to know the message has been received by a message broker or not.



There are various tools support lightweight messaging. Below are the some of the message brokers that is delivers the messages and consumes messages.

- RabbitMQ
- Apache Kafka
- Apache ActiveMQ

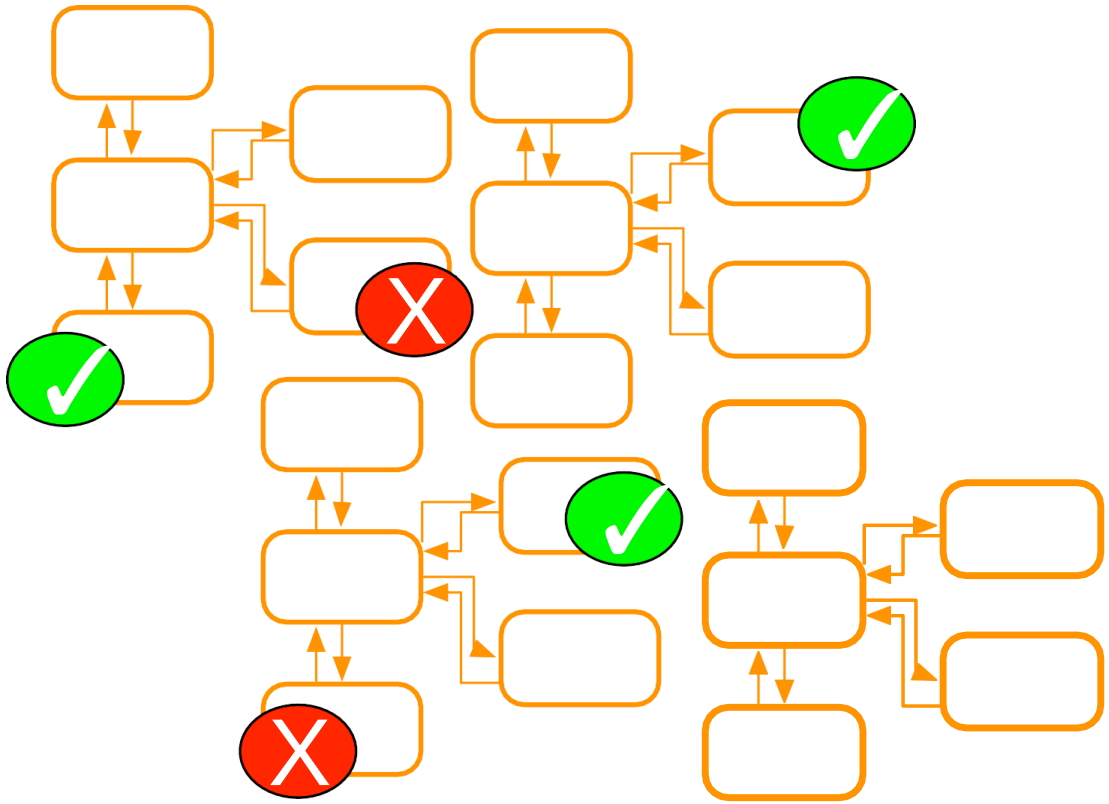
Benefits:

- Loose runtime coupling since it decouples the message sender from the consumer
- Improved availability since the message broker buffers messages until the consumer is able to process them
- Supports a variety of communication patterns including request/reply, notifications, request/async response, publish/subscribe, publish/async response etc.

Drawbacks:

Additional complexity of message broker, which must be highly available

Microservice Testing:



Tools to reduce testing

- Manual regression testing
- Time taken on testing integration
- Environment setup for testing

Tools to provide quick feedback

- Integration feedback on check in
- Continous Integration

Tools to provide quick deployment

- Pipeline to deployment
- Deployment ready status
- Automated deployment

- Reliable deployment
- Continuous Deployment

Why

- Distributed system
- Multiple instances of services
- Manual integration testing too time consuming
- Manual deployment time consuming and unreliable

Microservice Deployment Patterns:

We have designed application using microservice architecture pattern and application has set of services.

Each service is deployed as a set of service instances for throughput and availability.

Below are different deployment patterns available to deploy micro services.

- Multiple service instances per host
- Service instance per host
- Service instance per VM
- Service instance per Container
- Serverless deployment (Using AWS lambda)
- Service deployment platform

Circuit Breaker pattern:

In Microservice architecture, When one service synchronously invokes another there is always the possibility that the other service is unavailable or is exhibiting such high latency it is essentially unusable.

Precious resources such as threads might be consumed in the caller while waiting for the other service to respond.

This might lead to resource exhaustion, which would make the calling service unable to handle other requests.

The failure of one service can potentially cascade to other services throughout the application.

- A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker.
- When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately.
- After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

Use of the Circuit Breaker pattern can let a microservice continue operating when a related service fails, preventing the failure from cascading and giving the failing service time to recover.

Follow the below steps to configure the circuit backer:

1. Add the following dependency in pom.xml file

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

2.Add the below annotation in Spring boot main class

```
@EnableCircuitBreaker
```

3. Add below annotation at method level.

```
@HystrixCommand
```

Hystrix watches for failures in that method, and if failures reached a threshold (limit), Hystrix opens the circuit so that subsequent calls will automatically fail. Therefore, and while the circuit is open, Hystrix redirects calls to the fallback method.

4. Implement corresponding fallback method.

Example :

```
@Service
public class OrderService {

 @Autowired
 public RestTemplate restTemplate;

 @HystrixCommand(fallbackMethod = "defaultCustomer")
 public CustomerDetails getCustomerDetails(String customerId) {
    String url = "http://localhost:8081/customer/"+customerId;

    return restTemplate.getForObject(url, CustomerDetails.class);
 }

 public CustomerDetails defaultCustomer() {    // fallback method.
   CustomerDetails  customerDetails = new CustomerDetails();
   customerDetails.setStatus("FAIL");
   customerDetails.setMessage("No response from Customer service.");
   return customerDetails;
 }
}
```