# **The API gateway** has responsibilities to provide the application client with API, perform request routing, provide authentication, load balancing, monitoring, composition, caching, request throttling and protocol translation.

There are many reasons why we need an API gateway in the microservices architecture pattern, which are as given below:

* API gateway helps to stop exposing internal concerns to external clients.
* API gateway helps to provide additional security to your microservices.
* API gateway helps to merge the communication protocols.
* As you have studied, API gateway helps to decrease the complexity of microservices, eventually improving the efficiency of the application.
* API gateway helps separate the microservice API and other external APIs to virtualize the design requirements and testing.

Backend for Frontend: Aggregator. formatting data

**Service Mesh** is a network communication infrastructure which allows you to decouple and offload most of the application network functions from your service code. Hence when you do service-to-service communication, you don’t need to implement resilient communication patterns such as Circuit breakers, timeouts in your service’s code. Similarly, service mesh provides other functionalities such as service discovery, observability etc.

**Why Do You Need a Service Mesh?** Services within the Microservices-based architecture being modular in nature are difficult to manage. Whenever there is a service call from one Microservice to another, it’s abstruse (unmanaged) for the DevOps teams to infer or debug what’s happening inside the networked service calls.  
**What is a Service Mesh?** This is how a Service mesh such as an open-source project Istio service mesh can help resolve the issue. A service mesh is a configurable and dedicated infrastructure layer for managing service-to-service network communications within the cloud environment.

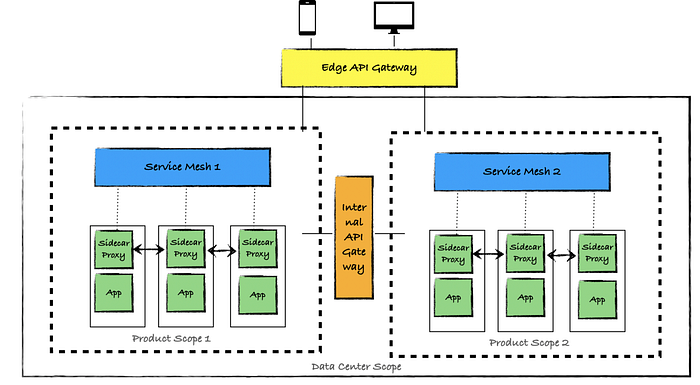
A service mesh consists of two elements,

- The Data Plane

- The Control Plane

Data plane controls and manages the actual forwarding of the traffic, the control plane provides the configuration and coordination. In the service mesh architecture, the data plane refers to the network proxies. These network proxies are deployed alongside each instance of a service that needs to communicate with the other services in the system. All service calls moving to and from a service go through these proxies. The proxy then applies rules of authentication, authorization, encryption, rate-limiting and load balancing, handles service discovery, and implements logging and tracing.

In order to minimize the additional latency, the proxy needs to be run on the same machine or in the same pod as the service, for which it was proxying so that they can communicate over the local host. This model is also known as sidecar deployment and hence named as “sidecar proxy”.



**Apache Camel:** Camel is an Open-Source **integration** framework that empowers you to quickly and easily integrate various systems consuming or producing data.

Apache Camel is a rule-based routing and mediation engine that provides a Java object-based implementation of the Enterprise Integration Patterns using an API (or declarative Java Domain Specific Language) to configure routing and mediation rules.

An API gateway stands between the user and internal application logic, while the service mesh is between the internal microservices.

<https://www.javainuse.com/camel>

Routes and routing engine are the central part of Camel. Routes contain the flow and logic of integration between different systems.

**basic Camel terminology and architecture**

First, we'll have a look at the core Camel concepts here:

* **Message** contains data which is being transferred to a route. Each message has a unique identifier and it's constructed out of a body, headers, and attachments
* **Exchange** is the container of a message and it is created when a message is received by a consumer during the routing process. Exchange allows different types of interaction between systems – it can define a one-way message or a request-response message
* **Endpoint** is a channel through which system can receive or send a message. It can refer to a web service URI, queue URI, file, email address, etc
* **Component** acts as an endpoint factory. To put it simply, components offer an interface to different technologies using the same approach and syntax. Camel already supports a lot of components in its DSLs for almost every possible technology, but it also gives the ability for writing custom components
* **Processor** is a simple Java interface which is used to add custom integration logic to a route. It contains a single process method used to preform custom business logic on a message received by a consumer

**What is an ESB?** The core concept of the ESB architecture is that you integrate different applications by putting a communication bus between them and then enable each application to talk to the bus. This decouples systems from each other, allowing them to communicate without dependency on or knowledge of other systems on the bus.

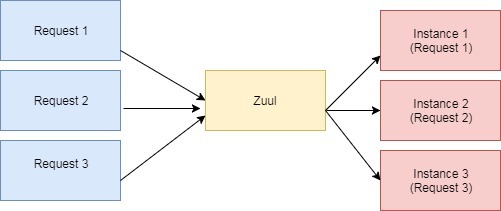
# A CsrfTokenRepository that persists the CSRF token in a cookie named "XSRF-TOKEN" and reads from the header "X-XSRF-TOKEN"

## Hazelcast is a distributed In-Memory Data Grid platform for Java. The architecture supports high scalability and data distribution in a clustered environment. It supports auto-discovery of nodes and intelligent synchronization.

## Load Balancing with Zuul

**When Zuul receives a request, it picks up one of the physical locations available and forwards requests to the actual service instance.** The whole process of caching the location of the service instances and forwarding the request to the actual location is provided out of the box with no additional configurations needed.

Here, we can see how Zuul is encapsulating three different instances of the same service:



Internally, Zuul uses Netflix Ribbon to look up for all instances of the service from the service discovery (Eureka Server).

# Zuul Proxy:

Zuul Server is a gateway application that handles all the requests and does the dynamic routing of microservice applications. The Zuul Server is also known as **Edge Server**.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

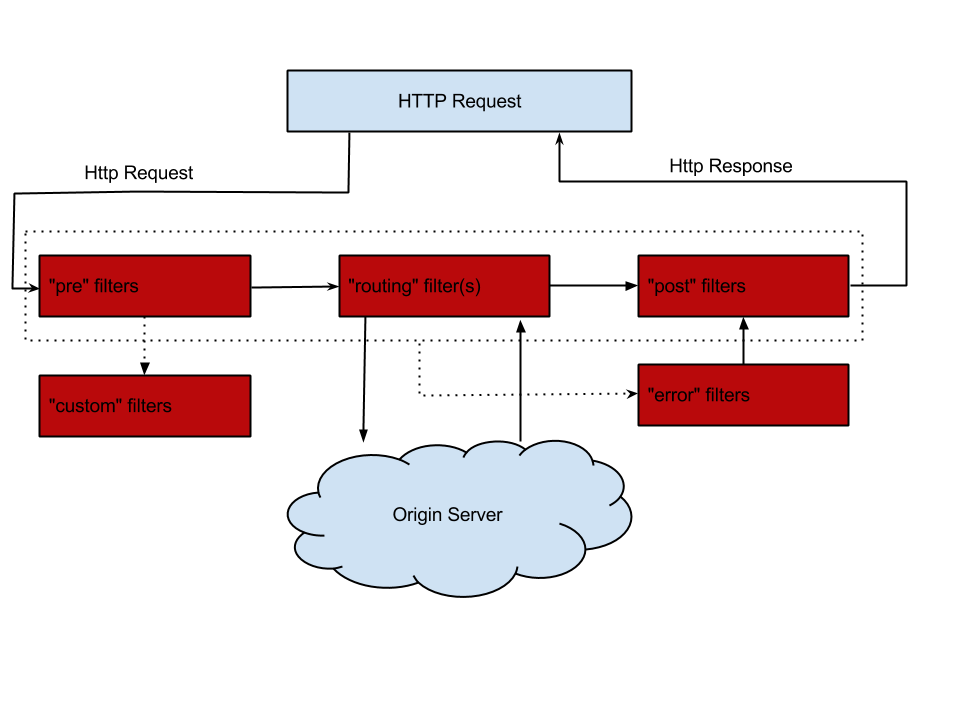
</dependency>

Application.properties

zuul.routes.products.path = /api/demo/\*\*

zuul.routes.products.url = http://localhost:8080/

The @EnableZuulProxy annotation is used to make your Spring Boot application act as a Zuul Proxy server.



Netflix [Ribbon](https://github.com/Netflix/ribbon) is an Inter Process Communication (IPC) cloud library. Ribbon primarily provides client-side load balancing algorithms.

Apart from the client-side load balancing algorithms, Ribbon provides also other features:

* **Service Discovery Integration** – Ribbon load balancers provide service discovery in dynamic environments like a cloud. Integration with Eureka and Netflix service discovery component is included in the ribbon library
* **Fault Tolerance** – the Ribbon API can dynamically determine whether the servers are up and running in a live environment and can detect those servers that are down
* **Configurable load-balancing rules** – Ribbon supports RoundRobinRule, AvailabilityFilteringRule, WeightedResponseTimeRule out of the box and also supports defining custom rules

Ribbon API works based on the concept called “Named Client”. While configuring Ribbon in our application configuration file we provide a name for the list of servers included for the load balancing.

Ribbon API enables us to configure the following components of the load balancer:

* **Rule** – Logic component which specifies the load balancing rule we are using in our application
* **Ping** – A Component which specifies the mechanism we use to determine the server’s availability in real-time
* ServerList – can be dynamic or static. In our case, we are using a static list of servers and hence we are defining them in the application configuration file directly

Notice how we used the *WeightedResponseTimeRule* rule to determine the server and *PingUrl* mechanism to determine the server’s availability in real-time. According to this rule, each server is given a weight according to its average response time, lesser the response time gives lesser the weight. This rule randomly selects a server where the possibility is determined by server’s weight. And the *PingUrl* will ping every URL to determine the server’s availability.

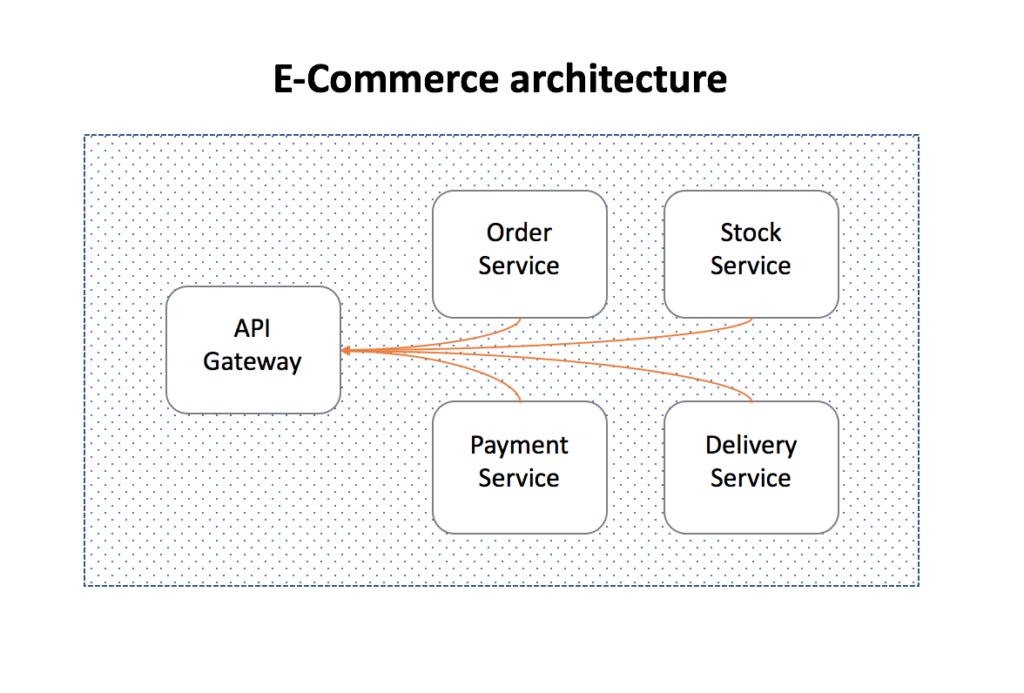
# Saga Pattern | How to Implement Business Transactions Using Microservices

One of the most powerful types of transactions is called a Two-Phase Commit, which is in summary when the commit of a first transaction depends on the completion of a second. It is useful especially when you have to update multiple entities at the same time, like confirming an order and updating the stock at once.

However, when you are working with microservices, for example, things get more complicated. Each service is a system apart with its own database, and you no longer can leverage the simplicity of local two-phase-commits to maintain the consistency of your whole system.

When you lose this ability, RDBMS becomes quite a bad choice for storage, as you could accomplish the same "single entity atomic transaction" but dozens of times faster by just using a NoSQL database like Couchbase. That is why the majority of companies working with microservices are also using NoSQL.

To exemplify this problem, consider the following high-level microservices architecture of an e-commerce system:



In the example above, one can't just place an order, charge the customer, update the stock, and send it to delivery all in a single ACID transaction. To execute this entire flow consistently, you would be required to create a distributed transaction.

We all know how difficult it is to implement anything distributed, and transactions, unfortunately, are no exception. Dealing with transient states, eventual consistency between services, isolations, and rollbacks are scenarios that should be considered during the design phase.

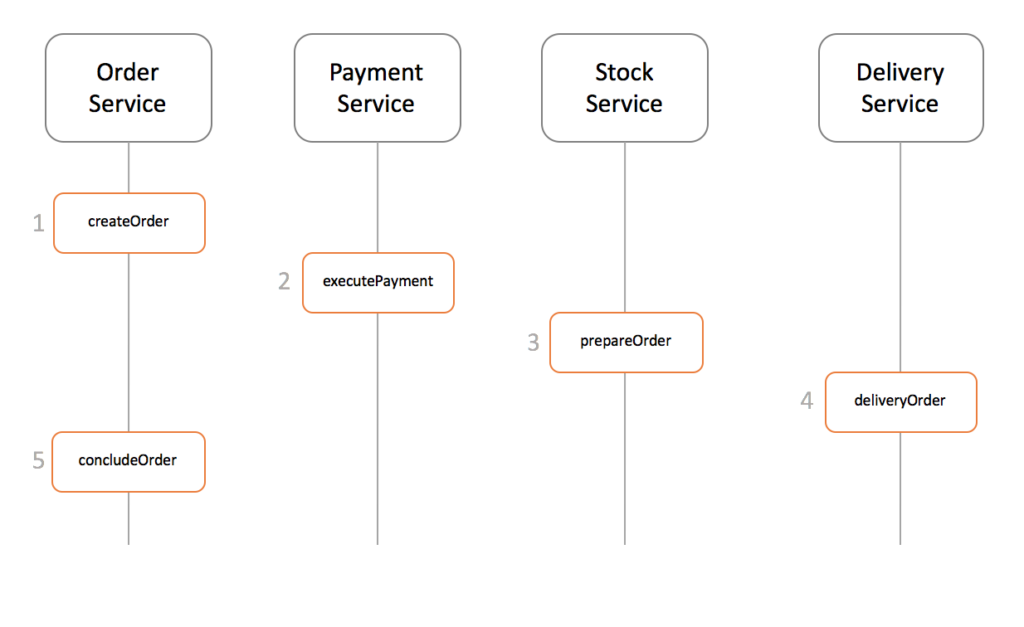
Fortunately, we have already come up with some good patterns for it, as we have been implementing distributed transactions for over twenty years now. The one that I would like to talk about today is called the Saga pattern.

**The Saga Pattern**

One of the most well-known patterns for **distributed transactions is called Saga**. The first paper about it [was published back in 1987,](https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf)and it has been a popular solution since then.

A Saga is a sequence of local transactions where each transaction updates data within a single service. The first transaction is initiated by an external request corresponding to the system operation, and then each subsequent step is triggered by the completion of the previous one.

Using our previous e-commerce example, in a very high-level design, a Saga implementation would look like the following:



There are a couple of different ways to implement a saga transaction, but the two most popular are:

1. **Events/Choreography:** When there is no central coordination, each service produces and listens to the other service's events and decides if an action should be taken or not.
2. **Command/Orchestration**: When a coordinator service is responsible for centralizing the saga's decision making and sequencing business logic.

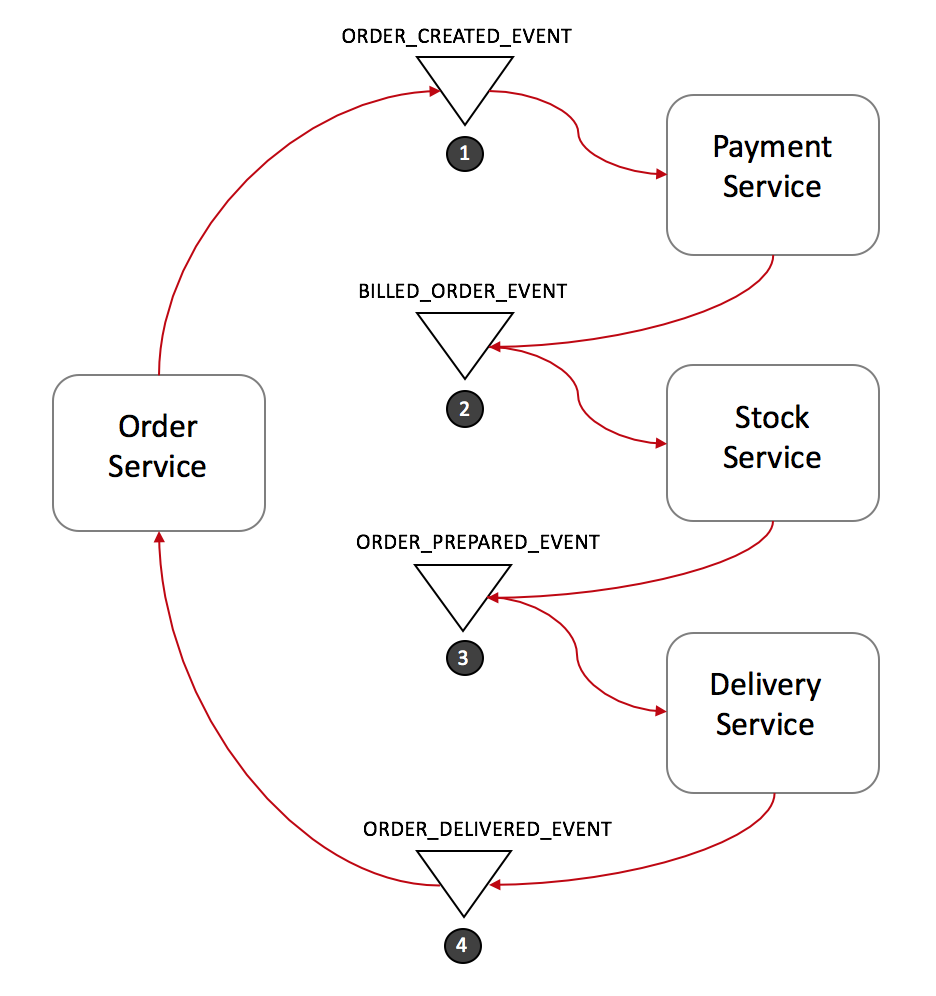
Let's go a little bit deeper into each implementation to understand how they work.

**Events/Choreography**

In the Events/Choreography approach, the first service executes a transaction and then publishes an event. This event is listened to by one or more services, which execute local transactions and publish (or don't publish) new events.

The distributed transaction ends when the last service executes its local transaction and does not publish any events, or the event published is not heard by any of the saga's participants.

Let's see how it would look in our e-commerce example:



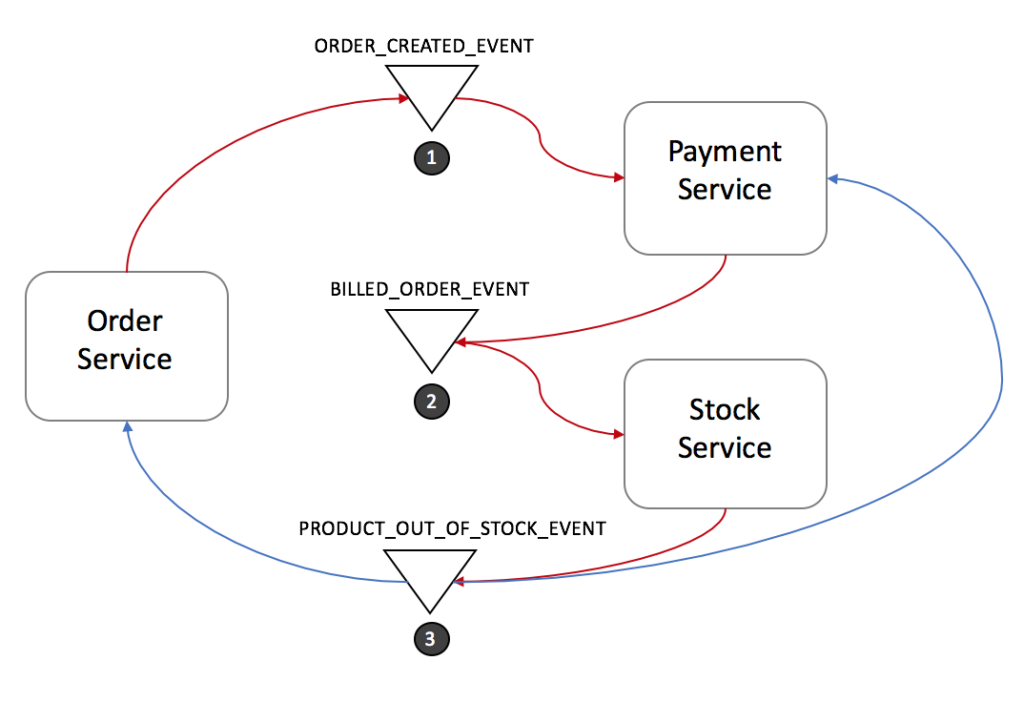
1. *Order Service* saves a new order, set the state as *pending* and publish an event called ***ORDER\_CREATED\_EVENT***.
2. The *Payment Service* listens to ***ORDER\_CREATED\_EVENT***, charge the client and publish the event ***BILLED\_ORDER\_EVENT***.
3. The *Stock Service* listens to ***BILLED\_ORDER\_EVENT***, update the stock, prepare the products bought in the order and publish ***ORDER\_PREPARED\_EVENT***.
4. *Delivery Service* listens to ***ORDER\_PREPARED\_EVENT*** and then pick up and deliver the product. At the end, it publishes an ***ORDER\_DELIVERED\_EVENT***
5. Finally, *Order Service* listens to ***ORDER\_DELIVERED\_EVENT*** and set the state of the order as concluded.

In the case above, if the state of the order needs to be tracked, Order Service could simply listen to all events and update its state.

**Rollbacks in distributed transactions**

Rolling back a distributed transaction does not come for free. Normally you have to implement another operation/transaction to compensate for what has been done before.

Suppose that Stock Service has failed during a transaction. Let's see what the rollback would look like:



1. *Stock Service* produces ***PRODUCT\_OUT\_OF\_STOCK\_EVENT***;
2. Both *Order Service* and *Payment Servic*e listen to the previous message:
   1. P*ayment Service* refund the client.
   2. *Order Service* set the order state as failed.

Note that it is **crucial to define a common shared ID** for each transaction, so whenever you throw an event, all listeners can know right away which transaction it refers to.

**Benefits and Drawbacks of Saga's Event/Choreography Design**

Events/Choreography is a natural way to implement Saga's pattern; it is simple, easy to understand, does not require much effort to build, and all participants are loosely coupled as they don't have direct knowledge of each other. If your transaction involves 2 to 4 steps, it might be a very good fit.

However, this approach can rapidly become confusing if you keep adding extra steps in your transaction, as it is difficult to track which services listen to which events. Moreover, it also might add a cyclic dependency between services, as they have to subscribe to one another's events.

Finally, testing would be tricky to implement using this design. In order to simulate the transaction behavior, you should have all services running.

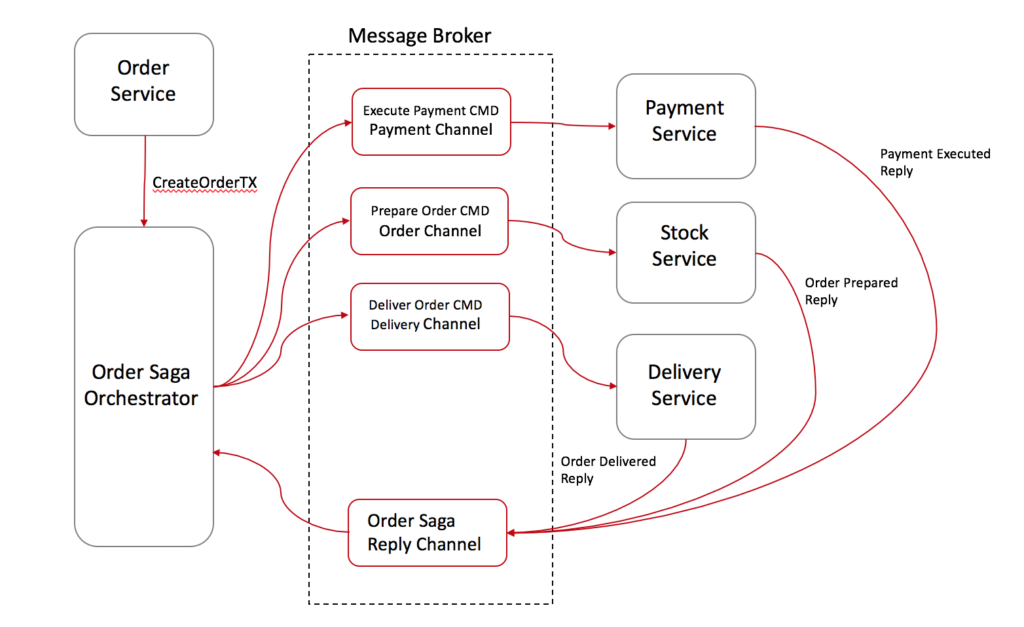
[In the next post](https://dzone.com/articles/saga-pattern-how-to-implement-business-transaction-1), I will explain how to address most of the problems with Saga's Events/Choreography approach using another Saga implementation called **Command/Orchestration**.

In the Previous post we saw some of the challenges of implementing distributed transactions and how to implement Saga's pattern using the Event/Choreography approach. In this article, let's talk about how to address some of its problems like complex transactions or cyclic dependencies of events by using another type of Saga's implementation called Command or Orchestration.

**Saga's Command/Orchestration Sequencing Logic**

In the orchestration approach, we define a new service with the sole responsibility of telling each participant what to do and when. The saga orchestrator communicates with each service in a command/reply style telling them what operation should be performed.

Let's see how it looks like using our previous e-commerce example:



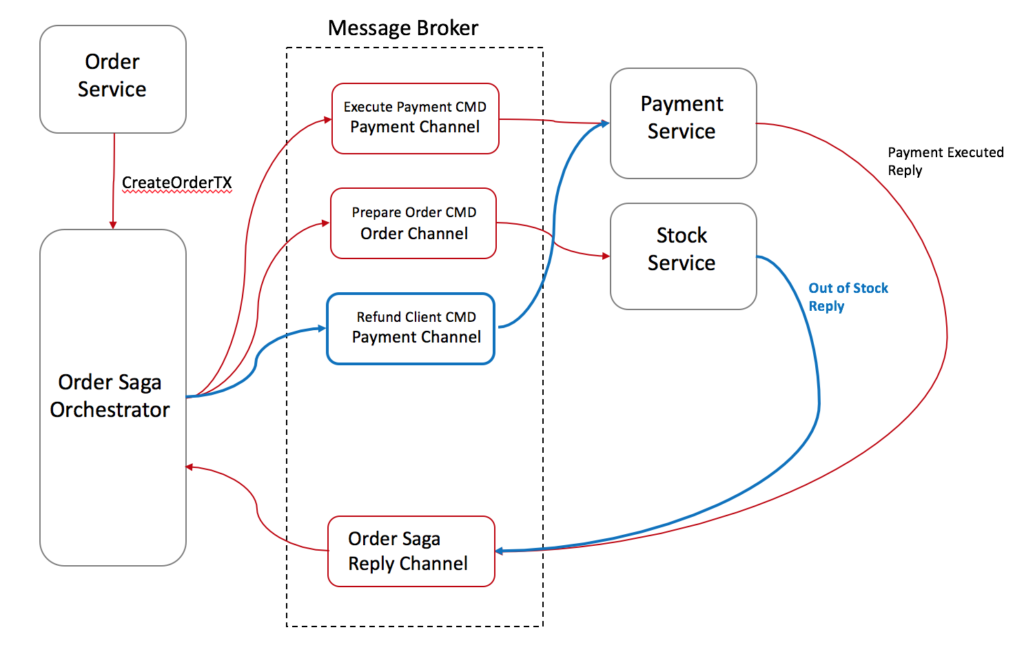
1. *Order Service* saves a pending order and asks Order Saga Orchestrator (OSO) to start a *create order transaction*.
2. *OSO* sends an ***Execute Payment*** command to *Payment Service*, and it replies with a ***Payment Executed*** message
3. *OSO* sends a ***Prepare Order*** command to Stock Service, and it replies with an ***Order Prepared*** message
4. *OSO* sends a ***Deliver Order*** command to Delivery Service, and it replies with an ***Order Delivered*** message

In the case above, Order Saga Orchestrator knows what is the flow needed to execute a "create order" transaction. If anything fails, it is also responsible for coordinating the rollback by sending commands to each participant to undo the previous operation.

A standard way to model a saga orchestrator is a State Machine where each transformation corresponds to a command or message. State machines are an excellent pattern to structure a well-defined behavior as they are easy to implement and particularly great for testing.

**Rolling Back in Saga's Command/Orchestration**

Rollbacks are a lot easier when you have an orchestrator to coordinate everything:



**Benefits and Drawbacks of Using Saga's Command/Orchestration Design**

Orchestration-based sagas have a variety of benefits:

* Avoid cyclic dependencies between services, as the saga orchestrator invokes the saga participants but the participants do not invoke the orchestrator
* Centralize the orchestration of the distributed transaction
* Reduce participant’s complexity as they only need to execute/reply commands.
* Easier to be implemented and tested
* The transaction complexity remains linear when new steps are added
* Rollbacks are easier to manage
* If you have a second transaction willing to change the same target object, you can easily put it on hold on the orchestrator until the first transaction ends.

However, this approach still has some drawbacks, one of them is the risk of concentrating too much logic in the orchestrator and ending up with an architecture where the smart orchestrator tells dumb services what to do.

Another downside of Saga's Orchestration-based is that it slightly increases your infrastructure complexity as you will need to manage an extra service.

**Saga Pattern Tips**

* **Create a Unique ID per Transaction(Spring Cloud Slueth)**

Having a unique identifier for each transaction is a common technique for traceability, but it also helps participants to have a standard way to request data from each other. By using a transaction Id, for instance, Delivery Service could ask Stock Service where to pick up the products and double check with the Payment Service if the order was paid.

* **Add the Reply Address Within the Command**

Instead of designing your participants to reply to a fixed address, consider sending the reply address within the message, this way you enable your participants to reply to multiple orchestrators.

* **Idempotent Operations**

If you are using queues for communication between services (like SQS, Kafka, RabbitMQ, etc.), I personally recommended you make your operations **Idempotent**. Most of those queues might deliver the same message twice.

It also might increase the fault tolerance of your service. Quite often a bug in a client might trigger/replay unwanted messages and mess up with your database.

* **Avoiding Synchronous Communications**

As the transaction goes, don't forget to add into the message all the data needed for each operation to be executed. The whole goal is to avoid synchronous calls between the services just to request more data. It will enable your services to execute their local transactions even when other services are offline.

The downside is that your orchestrator will be slightly more complex as you will need to manipulate the requests/responses of each step, so be aware of the tradeoffs.

**The Twelve Factors:** The Twelve-Factor App methodology provides best practices for building modern, scalable, and maintainable software-as-a-service applications. While it's not REST API-specific, it applies very well to REST API development, especially for microservices and cloud-native apps.

| **#** | **Factor** | **REST API Perspective** |
| --- | --- | --- |
| **1️⃣** | **Codebase** | **One codebase per REST API service; tracked in version control (e.g., Git).** |
| **2️⃣** | **Dependencies** | **Declare all dependencies (e.g., via Maven/Gradle for Java or package.json for Node.js). No global/system packages.** |
| **3️⃣** | **Config** | **Externalize config (DB URLs, API keys) via env vars, not hardcoded in code.** |
| **4️⃣** | **Backing Services** | **Treat DBs, caches, queues, and external APIs as attached services — accessed via config, not tightly coupled.** |
| **5️⃣** | **Build, Release, Run** | **Separate build (compile), release (env config), and run (deployment) phases. Enables repeatable deployments.** |
| **6️⃣** | **Processes** | **REST APIs run as stateless processes — no session data stored in memory; use external stores for state.** |
| **7️⃣** | **Port Binding** | **Expose services via self-contained HTTP servers, binding to a port (e.g., 8080).** |
| **8️⃣** | **Concurrency** | **Scale out via processes/threads or containers (e.g., Kubernetes pods), not by increasing in-app threads.** |
| **9️⃣** | **Disposability** | **REST APIs should start fast and shut down gracefully (important for scaling and updates).** |
| **🔟** | **Dev/Prod Parity** | **Keep dev, staging, and production environments as similar as possible to avoid deployment surprises.** |
| **1️⃣1️⃣** | **Logs** | **REST APIs should write logs to stdout/stderr. Log aggregation (e.g., ELK stack) is done by external systems.** |
| **1️⃣2️⃣** | **Admin Processes** | **Admin tasks (DB migrations, cleanup) run as one-off scripts, not built into the API itself.** |