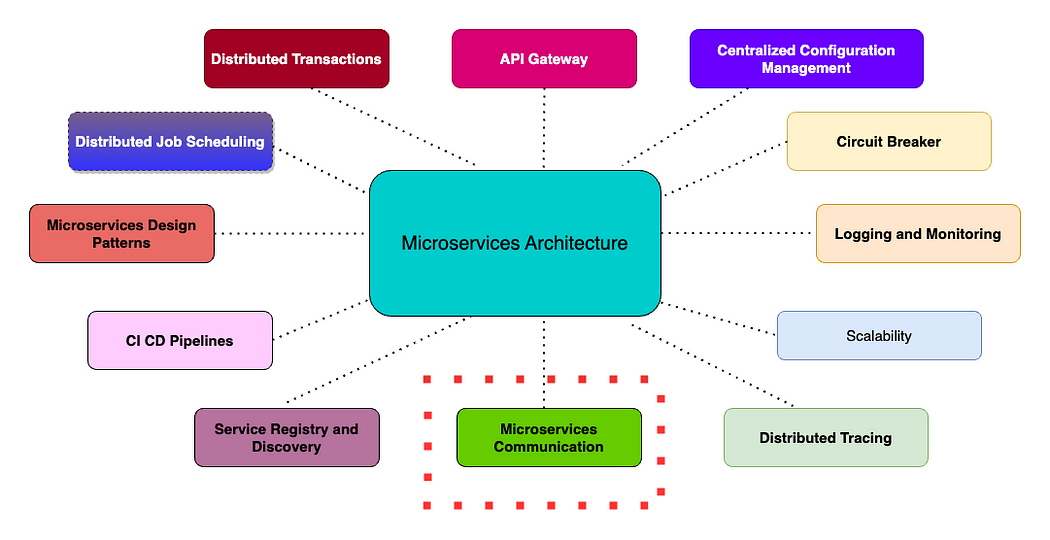


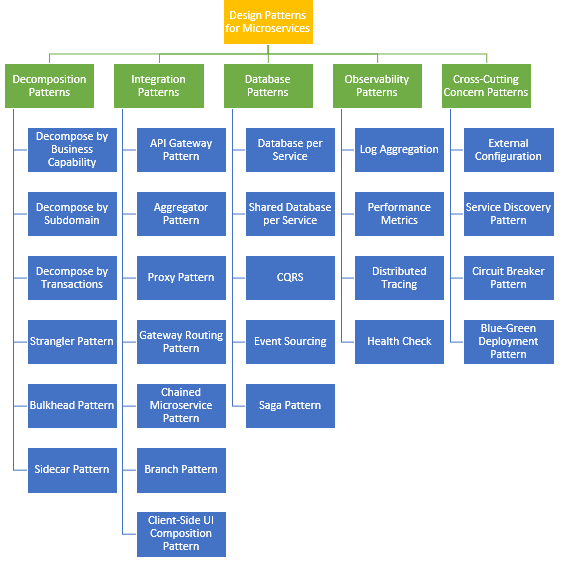
**What are microservices**

Microservices - also known as the [microservice architecture](https://microservices.io/patterns/microservices.html) - is an architectural style that structures an application as a collection of services that are:

* [Independently deployable](https://microservices.io/post/architecture/2022/05/04/microservice-architecture-essentials-deployability.html)
* [Loosely coupled](https://microservices.io/post/architecture/2023/03/28/microservice-architecture-essentials-loose-coupling.html)
* Organized around business capabilities
* Owned by a small team

While we are developing Microservice Architecture we need to consider below points-

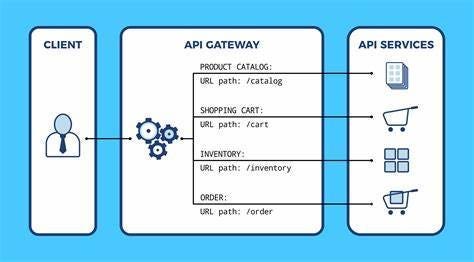




[Microservices design patterns](https://medium.com/javarevisited/top-10-microservice-design-patterns-for-experienced-developers-f4f5f782810e) have become increasingly popular due to their ability to improve software agility, scalability, resilience, and maintainability. [Microservices design patterns](https://medium.com/javarevisited/what-is-database-per-microservices-pattern-what-problem-does-it-solve-60b8c5478825) are a set of principles and best practices used to develop and maintain software systems that are composed of small, independently deployable services.In this article, we will discuss the important microservice design patterns that can be used to create robust and efficient [microservices architectures](https://medium.com/javarevisited/difference-between-microservices-and-monolithic-architecture-for-java-interviews-af525908c2d5).

**5 Important Microservices Design Patterns**

1. **API Gateway:**

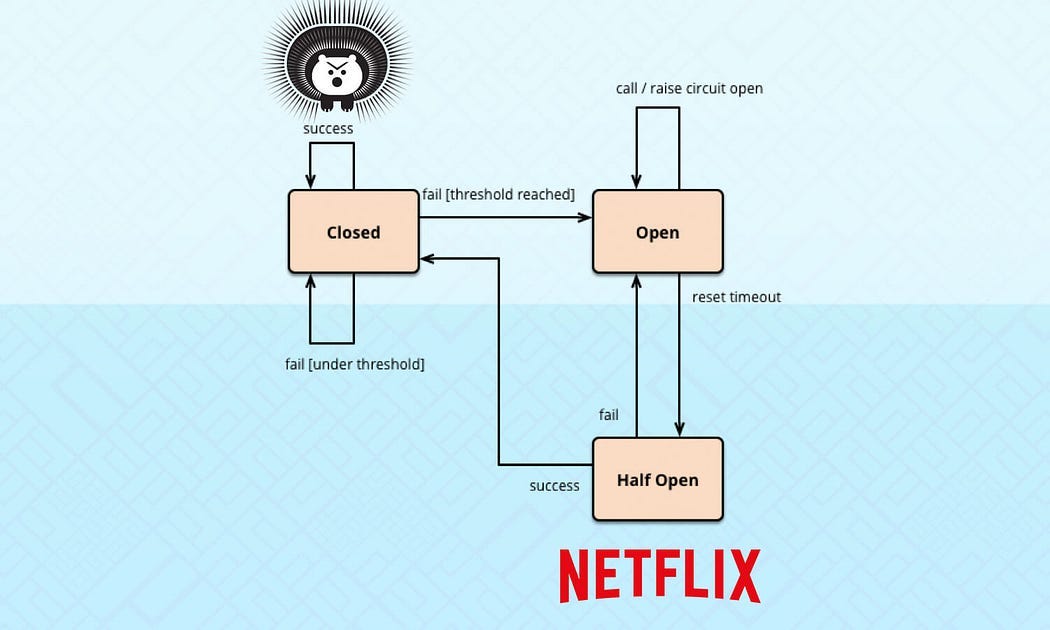
[[](https://www.java67.com/2023/04/3-what-is-api-gateway-design-pattern-in.html)](https://www.java67.com/2023/04/3-what-is-api-gateway-design-pattern-in.html)

* An [API Gateway](https://medium.com/javarevisited/how-to-build-a-microservice-ecosystem-with-api-gateway-cloud-config-and-zipkin-distributed-a783e00f1262) is a service that acts as a front-end for microservices. It receives requests from clients and routes them to the appropriate service.

**How API Gateway Pattern works:**

* The API Gateway pattern works by intercepting requests from clients and routing them to the appropriate service. When a client makes a request to the system, the request is first sent to the API Gateway. The API Gateway then checks if the request is authorized and if so, it routes the request to the appropriate service.
* The[API Gateway](https://medium.com/javarevisited/what-is-api-gateway-pattern-in-microservices-architecture-what-problem-does-it-solve-ebf75ae84698) can also perform other functions such as rate limiting, caching, and authentication. For example, it can limit the number of requests that a client can make to a service in a given time frame. It can also cache responses from services to reduce the load on the underlying services.

1. **Circuit Breaker:**

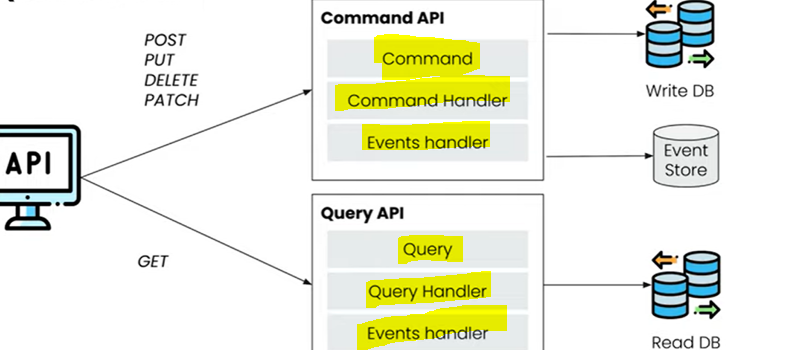
[[](https://www.java67.com/2023/04/what-is-circuit-breaker-design-pattern.html)](https://www.java67.com/2023/04/what-is-circuit-breaker-design-pattern.html)

* The [Circuit Breaker pattern](https://medium.com/javarevisited/what-is-circuit-breaker-design-pattern-in-microservices-java-spring-cloud-netflix-hystrix-example-f285929d7f68) is used to detect failures and prevent cascading failures in a distributed system.
* It operates like an electrical circuit breaker, which trips and breaks the circuit when there is an overload, preventing damage to the electrical equipment. Similarly, the Circuit Breaker pattern can trip and stop the flow of traffic to a failing system.

**How Circuit Breaker pattern works:**

* The [Circuit Breaker pattern](https://www.java67.com/2023/04/what-is-circuit-breaker-design-pattern.html) works by monitoring the health of a system component and taking action when it detects a failure. The Circuit Breaker has three states: closed, open, and half-open.
* Closed state: In the closed state, the Circuit Breaker allows traffic to flow normally to the system component. The Circuit Breaker monitors the response times and error rates of the system component.
* Open state: If the Circuit Breaker detects that the system component is failing, it trips and enters the open state. In the open state, the Circuit Breaker stops all traffic to the system component and redirects traffic to a backup component or returns an error message to the client.
* Half-open state: After a certain amount of time has passed, the Circuit Breaker enters the half-open state. In the half-open state, the Circuit Breaker allows a limited amount of traffic to flow to the system component. If the system component responds successfully to the traffic, the Circuit Breaker returns to the closed state. If the system component fails to respond successfully, the [Circuit Breaker returns](https://www.java67.com/2023/04/what-is-circuit-breaker-design-pattern.html) to the open state.

**3. CQRS (Command Query Responsibility Segregation)**

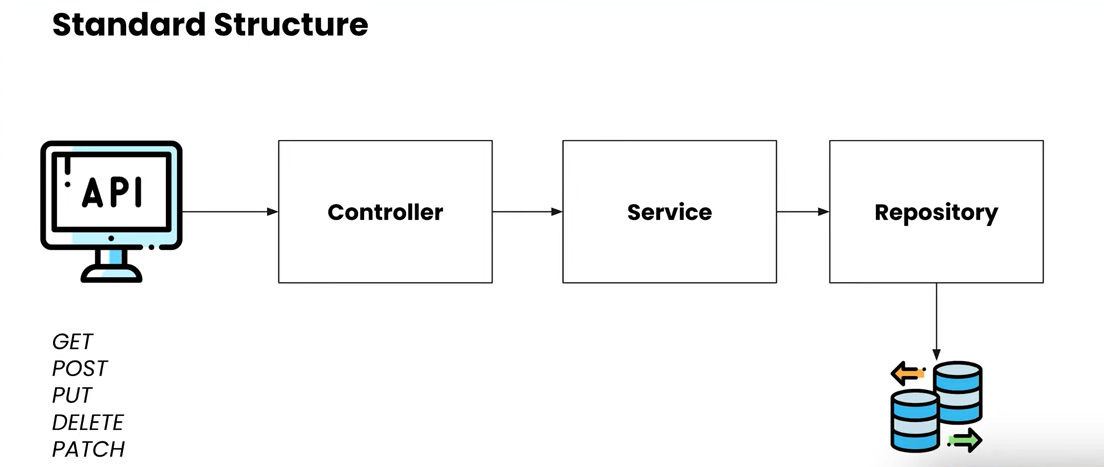


* In a traditional system, there is a single model that handles both read and write operations. This model is responsible for maintaining the state of the system, processing commands, and returning data to the client. However, as the system grows in complexity and size, it becomes difficult to scale and maintain.
* In order to overcome the above issue, [CQRS pattern](https://medium.com/javarevisited/what-is-cqrs-command-and-query-responsibility-segregation-pattern-7b1b38514edd) separates the read and write operations into two different models: the Command model and the Query model. The Command model is responsible for handling write operations and updating the state of the system. The Query model is responsible for handling read operations and returning data to the client.

**How CQRS pattern works:**

CQRS means **Command and Query Responsibility Segregation**. So, that mean we will be Segregating the Responsibility of a Command and a Query in a different way. To understand what a Command is and what a Query is let’s first understand our traditional flow or traditional architecture how we are building the application and how we can improve upon to implementing the CQRS. Now, one thing to understand is, **this CQRS we are going to implement using the event driven architecture.**

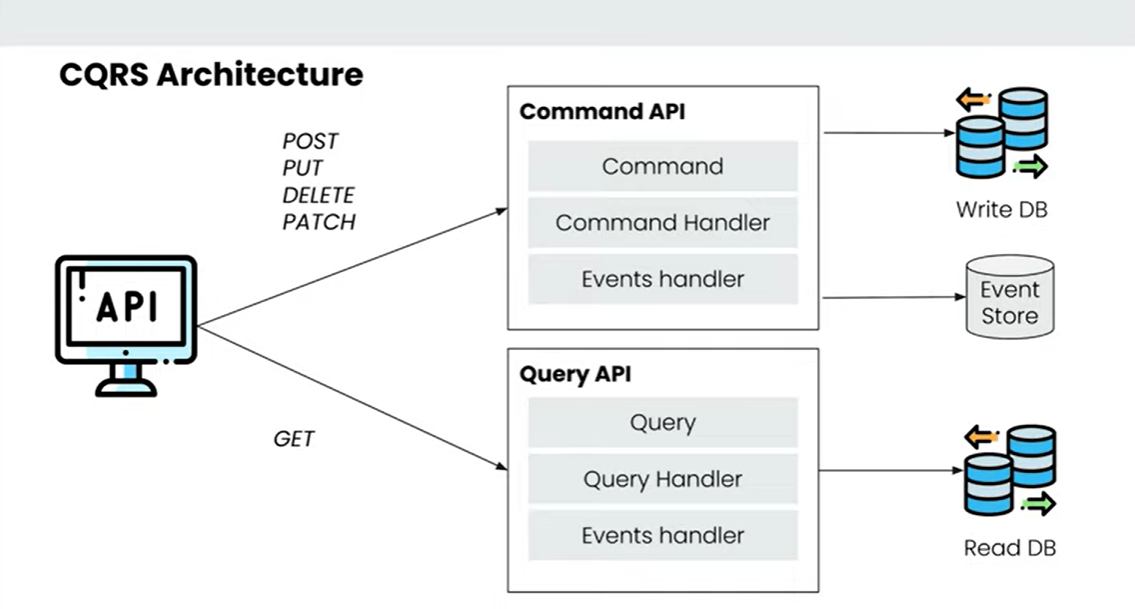
So, **To develop event driven architecture we will use the event sourcing to interact with each other**. So, first let’s understand the standard structure over here.



So, In the Standard Structure you might have seen multiple times like we will be creating the API, those API’s might be having the different HTTP Method Types **GET**, **POST**, **PUT**, **DELETE**, **PATCH**. So, you are creating the API and your API is controlled by your Controller Layer. So, that Controller Layer will be responsible for Traversing the Request to the each of the different Services that you have in your application. So, from the Controller your Request will be going to the Service Layer that Service Layer responsible for your Business Logic. So, whatever the business logic that you have for your application that you are building all those particular logics should be written in **Service Layer**. So, from the Service Layer you will be calling the **DAO Layer** or **Repository Layer** to save your data or to persist your data to the database. so, this is a very standard structure for creating a Spring MVC application or a standard Microservices as well.

But here let’s understand the CQRS Architecture, how it is different from the normal Architecture and what are the different things we are going to see over here.

Now the first thing we can see over here is we will be creating the API over here as we are creating the API in standard structure but the only difference is How and what are the different **HTTP** Method’s we are going to use. So, all the method that is to fetch the data i.e the **GET** method. We will be implementing the GET method to Fetch the data that particular will be implemented by the **Query API**.



This is the very basic level architecture we are seeing over here…

all the other methods that is **POST, PUT, DELET** and **PATCH** those are used to add the data, modify the data, and update the data that will be implemented by **Command API**. So, this is the 2 differences.

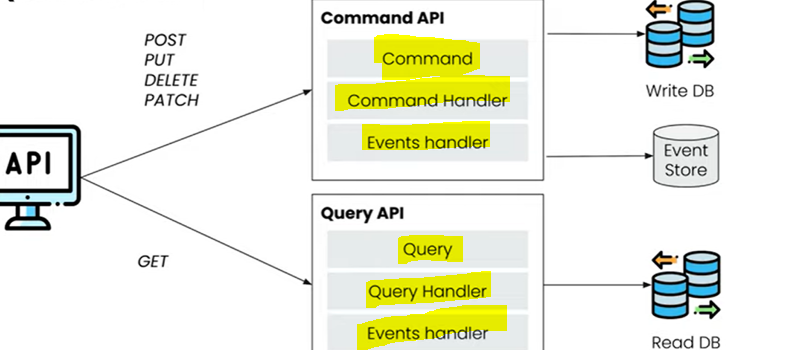
Whenever you are providing any Command to your Application to do any of the operation that will be handled by the **Command API** and whenever you want to Fetch any data that is already been added to your System to Fetch that you will be using **Query API**.

**Now, the differentiation of this Command API and Query API is the Core Concepts behind the CQRS.**

**i.e To Segregate the responsibilities of the Command’s and your Query’s that you have in your System.**

Ideally when you develop the System both of this particular command and query will be in different services so you can configure your system according to the need of the application. So, you are creating the Query API. you can tune the application to get the data and when you are using the Command API you can tune the application to store or update the data into a database.

So, for each and every Command API and Query API the main thing you are going to do is, for the Command API you will be providing the Command to execute any of the Command and for the Query API you will be providing the Queries to Fetch any of the data available. So, these are the 2 things that you will be providing in the System to do any of the operations over here.

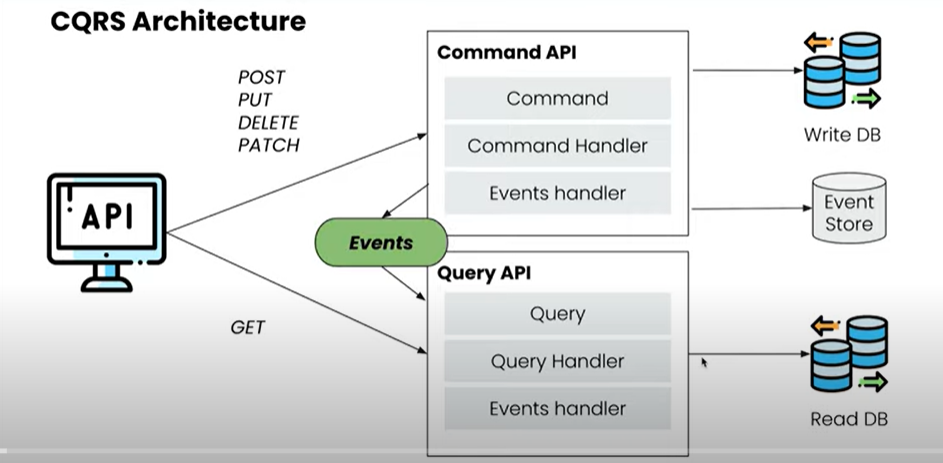


Now, for the **Command** to Handle all this particular command’s there will be a **Command Handler** that will be handling the command. and for the **Query** there will be a **Query Handler** that will be handling the queries.

Now this particular Command Handler also known as the **Aggregate.** when you will be developing the application will be creating the **Aggregate**, that particular Aggregate will be handling all the commands and for the queries we will be creating the **Projections**, that particular Projection will be handling your all queries. You can create the **Command Handler** classes as well and **Query Handler** classes as well. But you can also create the **Aggregate** and **Projection** as well.

So, I have seen multiple places where people have been using Aggregate and Projection and Command Handler and Query Handler as well and for Each and Every Command or Each and Every Query that you handle you will be creating the Events, so with that particular event your system can handle all the particular operations because we are going with the event-driven-architecture over here. All the particular events will be store into **Event Store**. Now for the **Command API** we will be storing data into the **Write Database** that means whatever the operations we do that particular database is configured for writing the data to that particular database. For the **Query API** we will be using the **Read Database** which is configured for reading the data from that particular database.

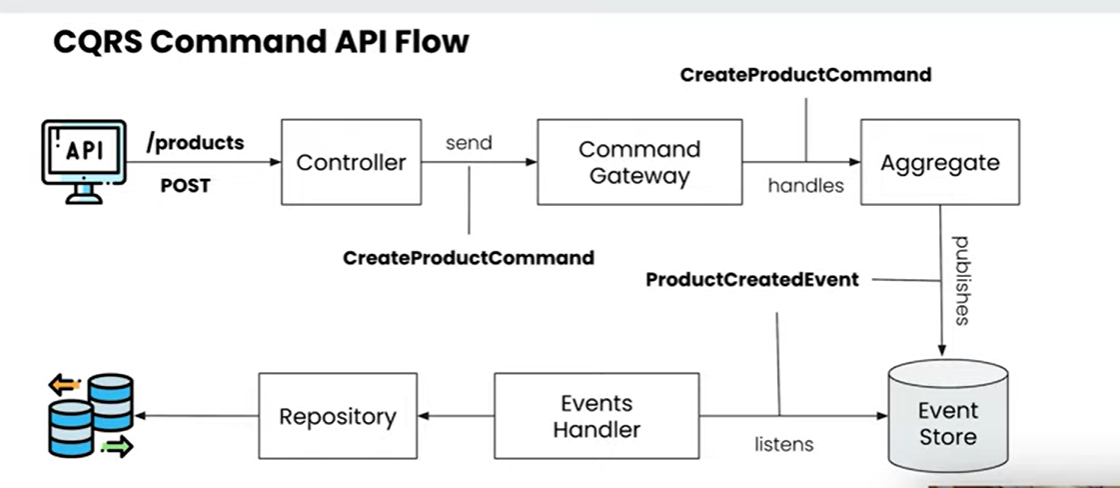
So, the question is whenever you store data into Write Database using the Command API how that particular data comes to the Read Database. so, those data come using the Events as well so for those type of operations also we will be creating the **Events** to store the data from the **Write DB** to the **Read DB**.



Now let’s see How to Implement the CQRS using the **Spring Boot** and **AXON Server**.

AXON Server provides the ability in this Spring Boot Application to handle all our different events and commands and store all our particular data in our Event Store. So, what we will be doing is we will be using the AXON server as well. So, that particular AXON server will store all our data and we will be using the AXON framework to interact with the particular AXON Server.

**Let’s understand the Data Flow for our Command API the similar thing will be available for the Query API as well…**



So, this is a CQRS command API flow where we have taken one example to save one of our products. So, we have a Product Service available, and we are going to save the product using the **POST** Request. So, here u can see that we have an API **/product,** and we are doing the **POST** request and we are having one **Controller,** this particular controller will be handling all our API Request, and based on the particular API’s that we are going to call based on the different **HTTP’s** methods. We will be sending the **command** to our **Command Gateway**. so, what is the command over here you can see that I have created a command called **CreateProductCommand** so, I am going to create any product using the **POST** method over here. So, I have added the command **CreateProductCommand.** you can create multiple commands like **UpdateProductCommand**, **DeleteProductCommand** based on the requirements and those particular commandsI am sending to the **Command Gateway.** I am creating **Aggregate** as well; **Aggregate** is nothing but a **Command Handler** over here which will handle all the commands which we are passing. So, this particular Aggregate will handle this particular command **CreateProductCommand** and whatever the operation like we have to do like validations and all those things we can do. We can create the **Event** like **ProductCreatedEvent**. So, if whatever the data we are passing over here if I think everything is ok then I can create the Event **ProductCreatedEvent** and that particular event will be store to an **Event Store**. We will be creating one **Event Handler** and that particular **Event Handler** will listens all of the Events which has been created and stored into **Event Store**. It will try to listen all those events and it will complete all those events. So, in this particular **Event Handler** what this particular Event Handler doing is it is trying to listen this particular **ProductCreatedEvent** over here. So, whenever a new event **ProductCreatedEvent** created and stored into **Event Store** it will consume that and it will listen that particular eventand this particular Event Handler will store that particular data into our database using our **Repository** so, this will be our **Write Database**.

So, this is the entire flow **event-driven-architecture** flow for our **command** for a CQRS this is for our Command parts.

So, this is the flow which we are going to use to create the different **Command** and create the different **Events**. according to the particular **Commands** handle those particular **Events.**

So, for the **Query API** this would be very similar there would be one **Controller** that will handle the particular API instead of **Command Gateway** there would be a **Query Gateway** that will handle those **Queries** and instead of **Aggregation** there would be a **Projection** that will handle those particular Projection and same it will have an **Event Handler** which will handle the events or we can directly send the data from the **Projection** as well.

So, we are going to create a Particular Product Service to create a Product and get the data from the particular products. So, this is the same flow we are going to create one Controller, Aggregate, Event Handler and Repository for saving the products.

Before start to do the Coding, let’s first understand what is **Event Store** over here… so, let’s go to the **AXON Server Website** and from there we can understand the Event Store.

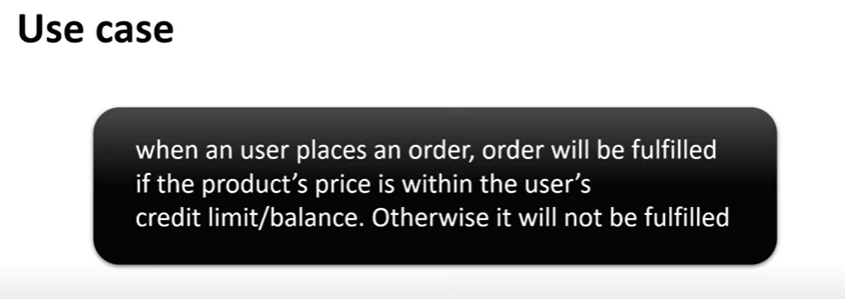
**4. Saga Pattern**

* Saga design pattern is a technique used in distributed systems to maintain consistency across multiple transactions involving multiple services. It is used to manage distributed transactions across multiple services in a way that ensures data consistency.
* In a distributed system, it’s common for a single transaction to involve multiple services. For example, in an e-commerce system, a single order may require updates to the inventory service, payment service, and shipping service. If any of these services fail, it can lead to inconsistencies in the data. The Saga pattern helps to ensure that the transaction is completed successfully or rolled back if any errors occur.
* The Saga pattern works by breaking a single transaction into multiple smaller transactions, also known as “compensating transactions.” These compensating transactions are executed in a particular order and are designed to undo the effects of the previous transaction if any errors occur. The order of the compensating transactions is crucial to ensure that the system remains consistent.

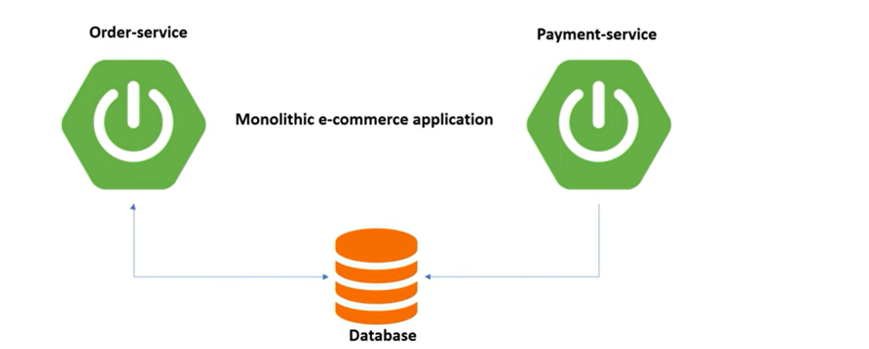
**There are two ways of coordination sagas:**

* **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

**SAGA Choreography**



let's try to map this use case with monolithic approach. Then we'll see how difficult it could be in dealing with transaction in the distributed microservices system. Okay. So, if you can see this diagram, we have order service and payment service and both the service pointing to the single database.

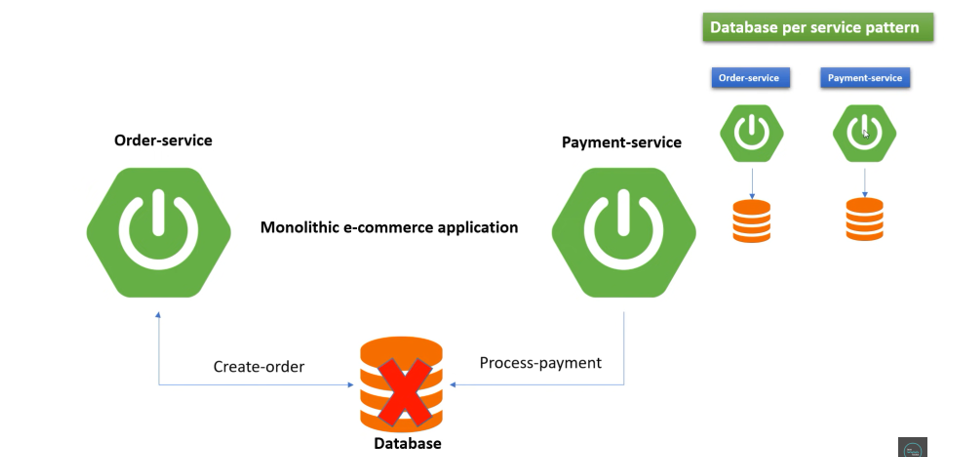


**Note:**~~- In Spring Boot, when @Transactional annotation is used, Spring Boot implicitly creates a proxy that will be creating a connection to the database. A transaction will be started and committed after the code has been executed errorless. Otherwise, it will roll back the changes if an exception occurred.~~

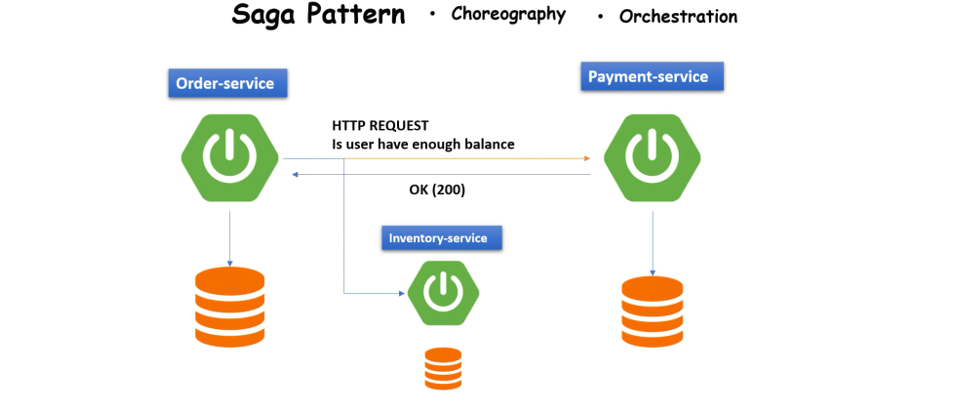
So this is very easy to implement in a monolithic application. The entire workflow can be considered as a single transaction because they both the application point to the single database. Right. And also it is easy to commit rollback when everything is in a single database. Let's say for example, order service send request to create a order.

Now again payment service will process the payment. So we call the process payment. Okay. In case there is some failure while processing the payment, then the entire data will be rolled back. Not a single data will be inserted to NADB. That is how the transaction handle in monolithic.

And the Spring framework provided enough feature to handle the transaction in case of monolithic application. But microservices does not follow this principle. Microservices basically follow the pattern called database per service where order service have its own database. Similarly, payment service have its own database. So if you observe, there is two microservices and they are pointing to two different database. If it is a single database, then the approach will be same as monolithic.



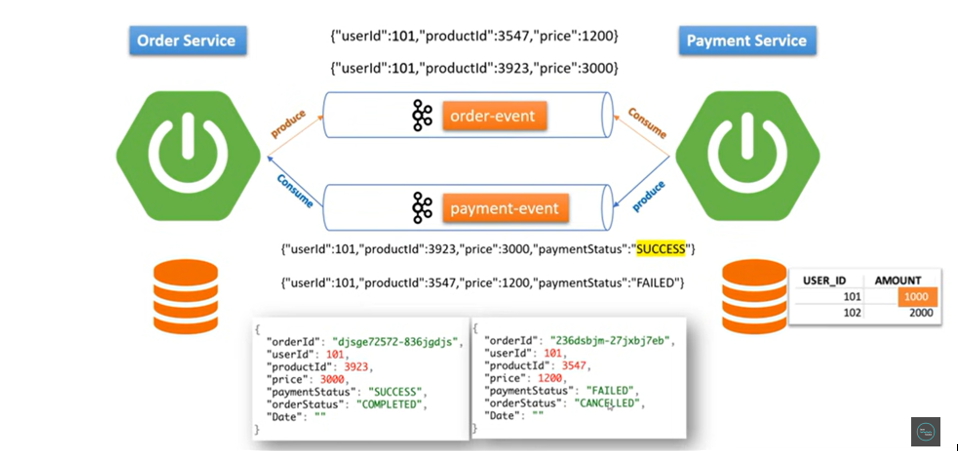
Now let's understand how we can handle in this microservices pattern where it already follows database per service pattern, how we can handle the transaction. So if you can see this diagram, order service has its own database, which is responsible for managing the order. Similarly, we have payment service has its own database, which is responsible for managing the payments. At the very first step, order service will receive the request from the user. Then again, order service will redirect that request to the payment service to validate whether that user have enough balance or not to check in the payment service. Now payment service receives that request and validate that user balance in DB. If he found that user have enough balance to purchase that order, he will deduct the amount from the DB and send OK response to the order service. Once order service receives this OK status, immediately he will complete the order. In case order service didn't receive OK status from the payment service, then we'll cancel the order. So this workflow is straightforward and very simple. But what if in future you want to introduce another service called inventory service who has his own database. So in that case, to check the product availability in inventory service, again order service need to send another HTTP request to check the product availability in inventory service, which is not a good approach to design the code. Unnecessarily for a simple requirement, we are firing n number of HTTP requests. So in future again, you just want to introduce another service from order service, you need to fire HTTP request to that particular service. And another problem I can say, order service always assume this payment service and inventory service is up and running. In case order service send request and payment service is not available that time, then it will be impact on your revenue, isn't it?



**So, in this approach, we found two disadvantages**.

1. One, you need to fire n number of HTTP requests.
2. second, it will impact on your revenue if there is an application downtime, right?

So, to overcome this issue, microservices developer provided a pattern called saga pattern. And we can implement this saga pattern with two approaches. One is **choreography approach**, and another is **orchestration approach**. So, in this tutorial, we'll be discussing the choreography-based approach by using event sourcing. So, there will be no HTTP request call, we just need to go with the event sourcing approach with choreography. Okay.



So now let's deep dive into choreography saga pattern and its architecture. As you understand, there is two service order service and payment service, and both of their own independent database. Okay, so once order service will receive the request from user, immediately he will produce an event to the Kafka topic and the name of topic is order event. Okay. So immediately, once he publishes the event to the order event topic, payment service will consume that event. And again, payment service will take that request and validate in user balance table to check whether that user have enough balance to purchase that order.

If he found that user have enough balance to purchase that order, immediately payment service will publish another event to the topic and the name of the topic is payment event. Now to this payment event topic, again, order service will consume that event. Okay. If order service found that response as a payment completed, then order service will complete that order. Otherwise, it will cancel the order. So, if you observe in this choreography pattern, there will be no HTTP request call. Right. Initially, order service publishes the event to the order event topic and payment service consume the event from the same topic. Then once payment service validates the request, again he publish the event to payment event topic and order service consume the event from the same topic. Right. So this approach looks good. Now let's try to sync this approach with one example.

For example, let's say in payment service, we have two users, user ID 101, 102 and they are having balance 4000 and 2000. Okay. So let's say order service just publish an event with user ID 101, product ID something and price is 3000. He just publish the event with this order event topic and immediately payment service will consume that event. He can see the entire message and then payment service will take this request and check in user balance database. User 1 having balance 4000 and he's trying to purchase something 3000. It means he has enough balance to purchase that order. So, payment service will immediately allow him to purchase the order and will deduct the amount. Now the rest amount is pending for user 101 is 1000. So immediately payment service after update the or after deduct the amount, he will publish an event to this payment event topic with the payment status success or completed something like that. Once order service will consume that event with payment status success or complete, we'll simply complete the order and the user will get the message something like this. Now for example, let's say user ID 101 again want to purchase something. So again, user service or order service send the request user ID 101 and product ID something different and price is 1200. Now payment service again will listen to the same topic and he can view this message or he can view this JSON. So payment service will take this request and will go to the user balance table and check that user ID 101 having the amount 1000 and he's trying to purchase something beyond the amount. It means it exceeds the limit. So, what payment service will do immediately will publish an event to the payment event topic with status failed. Now once order service will listen to that payment event topic or once order service consume that event with payment status failed, immediately order service will cancel that transaction or immediately order service will cancel that particular order and user will get a response like this. Okay, user ID this payment status failed, and order status is cancelled. So, this is how saga choreography pattern works. We'll try to implement the same replica of example.

We're going to use spring

**web flux,**

**Apache Kafka,**

**Spring Cloud Stream,**

**Java 8**

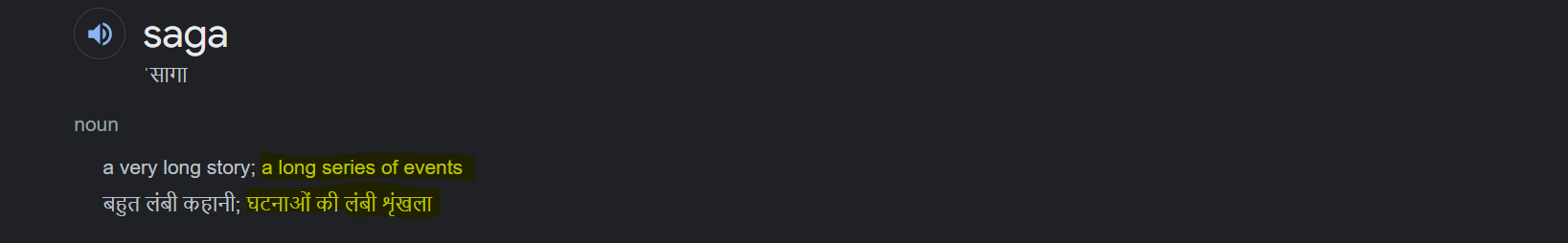
and also, we'll use multi module project so that we can reuse the component or classes.

**Saga Orchestrator**

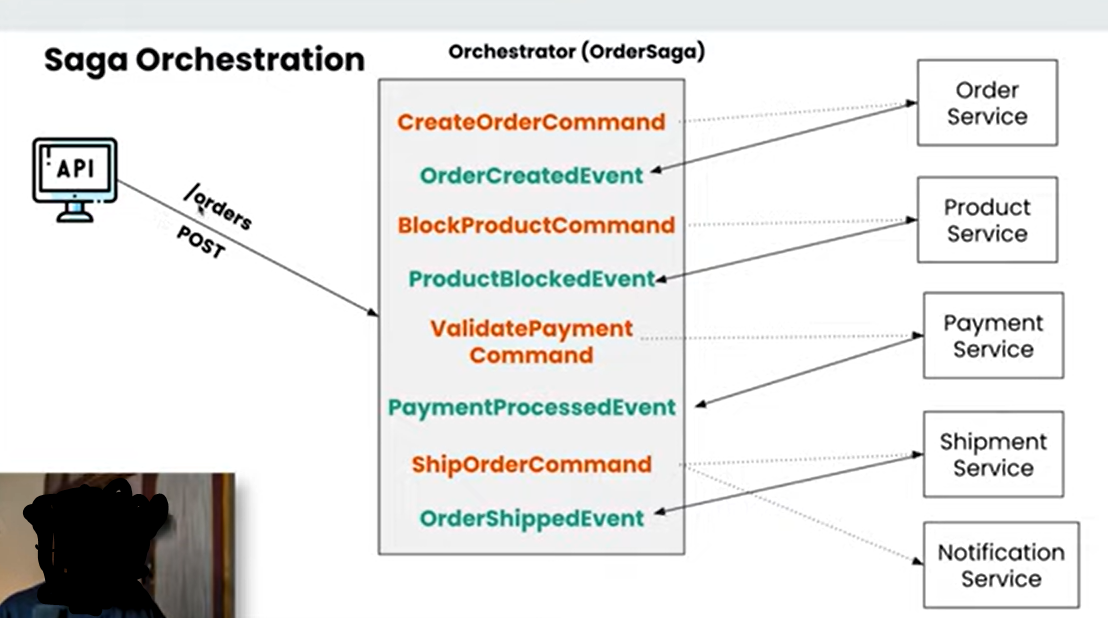
**First let’s understand What is Saga design pattern and why we need a Saga design pattern.**

**The Saga Design Pattern is a way to manage Data consistency across microservices in a distributed transaction scenario.**

So, there are multiple way’s can be handled and there is a one way that is a Saga Pattern. In saga also there are 2 things **Choreography Pattern** and **Orchestration Pattern**. In this tutorial we will learn more about the Saga Orchestrator. It’s simple thing when you have a transaction that span across the multiple services and how to handle that particular transaction will be using any of the design pattern of the microservices to handle those transaction.



**Order Saga -**



So, let’s understand the Saga Orchestration. So, here you can see that we have an API that is **POST** request that is doing the **/orders**. So, we are placing the order over here. In this Saga Orchestration pattern there would be one Orchestrator that will handle your entire saga that will handle everything that is been happening in your application.

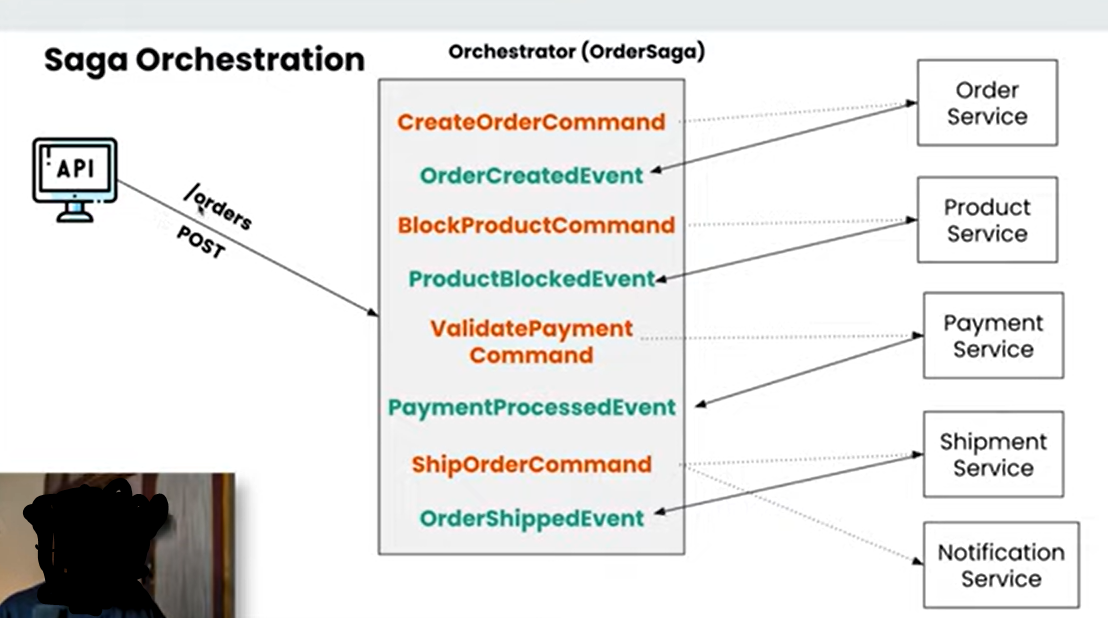
So, I am placing one order so that is one scenario. So, everything related to that particular order all the steps included in that will be handled by one orchestrator that I have defined here as an **OrderSaga**. So, this particular things in the microservices we are implementing using the **event-driven-architecture**.

**---------------------------------------- Brief about the Event- Driven- Architecture-------------------------------------**

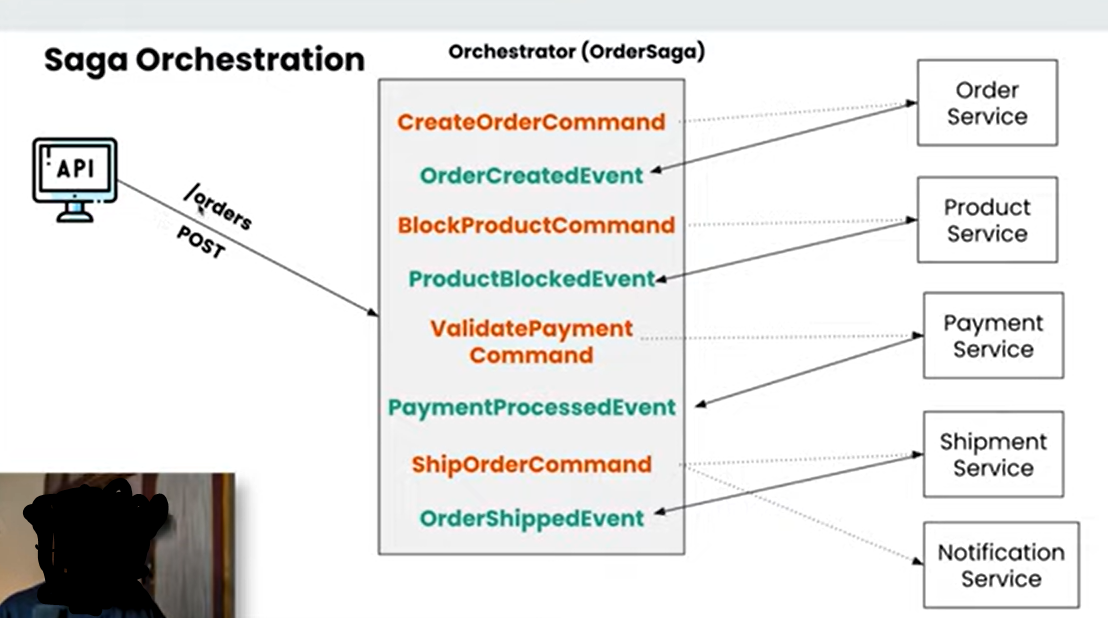
**We will be creating events and commands for every operation that we are going to perform**.

let’s come back to the Saga Orchestrator. So here you can see that there are different services available. So, these all are the different microservices that is **Order Service, Product Service, Payment Service, Shipment Service** and **Notification Service** and all.

Below one is our **Orchestrator** that will handle all particular services and which particular services to be called for the different operations. So, when you place a particular order at that particular time the Orchestrator is creating a command like **CreateOrderCommand** that means it is giving a command like I got the request to create the order and I am giving you the order to create the order so, this is particular **CreateOrderCommand**. So, **CreateOrderCommand** will be consumed by the **Order Service.**



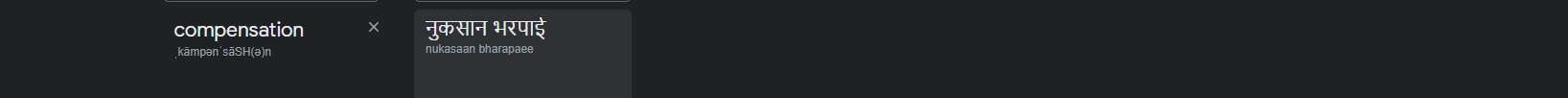
This particular **OrderService** will have the **CommandHandler** for that particular order. From that particular **CommandHandler** this particular command will be handled and at that particular time all the validations and all the other things will be performed and after that once the particular validation and other things performed that particular order service will create the event. so, that particular event will be **OrderCreatedEvent**. At that particular time **OrderCreatedEvent** will be consumed by **OrderSaga** as well plus you **OrderService** as well. So, both the things will consume that particular event. What **OrderService** will do, **OrderService** will store that particular order and it will say that I got the order. So, that particular data will be store to database for order service and this particular event “**OrderCreatedEvent**” will be consumed by the **OrderSaga** and this particular **OrderSaga** will know that ok I got this particular event that means the order detail has been stored in the database. now after that I need to lock that particular product. Bcz suppose that I have the quantity of 5 in my stock and when one particular order has been done, I need to block that particular order. So, it will create a new command to block that particular product. This particular “**BlockProductCommand**” command will be consumed by the **ProductService**. and that particular **ProductService** will be remove that particular quantity from that product table and then it will create a new event called “**ProductBlockedEvent**” so, **OrderSaga** will get to know ok this product is also blocked now. now let’s go to the next step that is the **To Validate the Payments.** So, **OrderSaga** will create a new command that is **ValidatePaymentCommand** so this **ValidatePaymentCommand** will be consumed by **PaymentService**. This **PaymentService** again in return create an event that is **PaymentProcessedEvent** again this event **PaymentProcessedEvent** will be consumed by both the services **OrderSaga** and **PaymentService** both**. PaymentService** again perform the payment processing and store the details into database. and **OrderSaga** once consumed this particular **PaymentProcessedEvent** it will go to the next steps to give the next command that is to **ShipOrderCommand**. So, it will create a **ShipOrderCommand** and that particular **ShipOrderCommand** will be consumed and will be handled by the **ShipmentService** and this particular **ShipmentService** will create the particular event that is **OrderShippedEvent** and this particular **ShipOrderCommand** will be consumed by the **NotificationService** as well.so, **NotificationService** what it will do is it will also consume that particular event and it will notify the user that we got your order, and we will ship that particular order. And at that particular end (--**OrderShippedEvent--)** like order is shipped now and at that particular time entire saga will be completed. So, until and unless that entire saga is been completed your order won’t be processed. If there is any error that particular entire transaction can be revoke. So, this is the happy scenario now let’s understand how the **transaction** are---

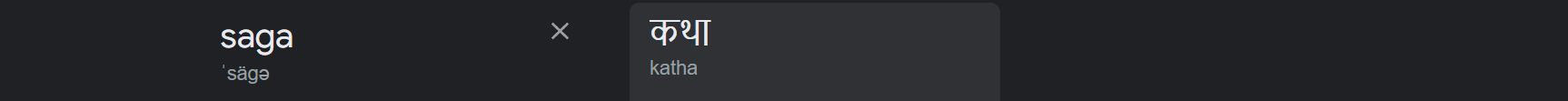


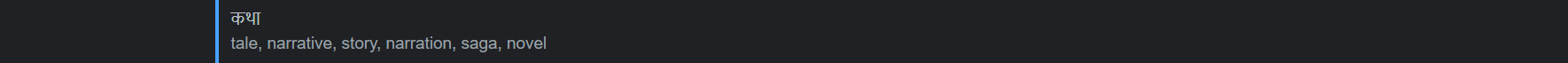
**Saga Transaction**



So, we can see that the saga transaction will be defined in this way. There will be multiple transactions available and as it is span across the different microservices you don’t have that ability to revoke that particular transaction. So, for all the transaction you do in all the different microservices in that particular project you have to have the compensating transaction to perform the revoking operation.

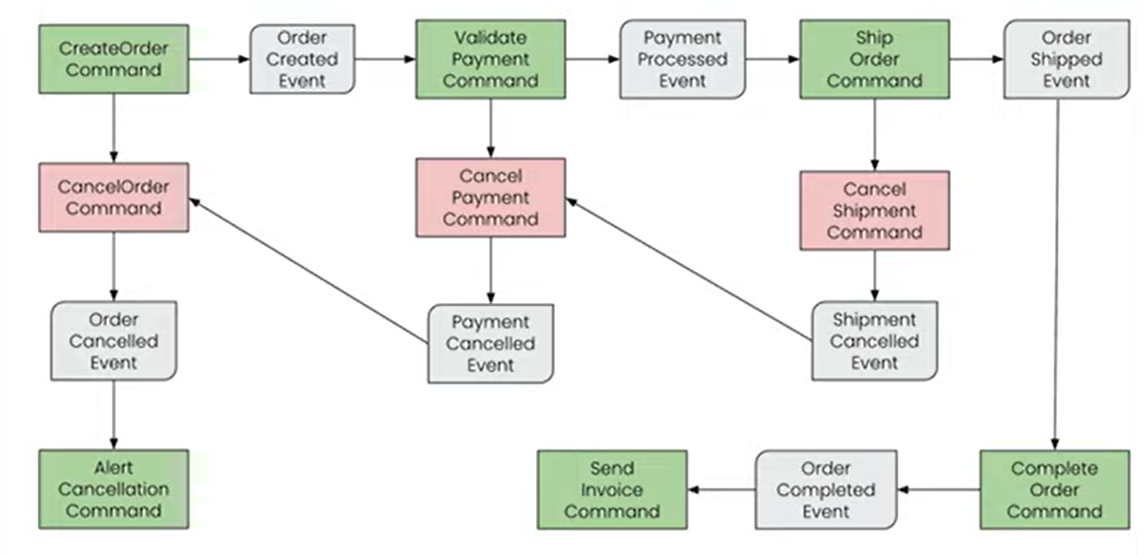






So, suppose you created the order and after that you created the payment and your particular payment service failed so at that time you have to have the compensating transaction for your order as well. That order whatever that stored has to be cancelled. So here you can see T1, T2, T3……upto Tn and the compensating transaction should be C1, C, C3 ……upto Cn-1. Now if you have gone upto the 3rd steps like T1, T2 and T3 and some thing happens at this particular point then at this particular point you should be having the 2 compensating transaction because you are at the 3rd place that should be C(3-1) and C(3-2) i.e C2 and C1. Just remember that there would be transaction and there would be compensating transaction for that because **we can’t handle the atomicity directly in microservices.**

**Workflow to Build Application + Transactions + Compensating Transaction**

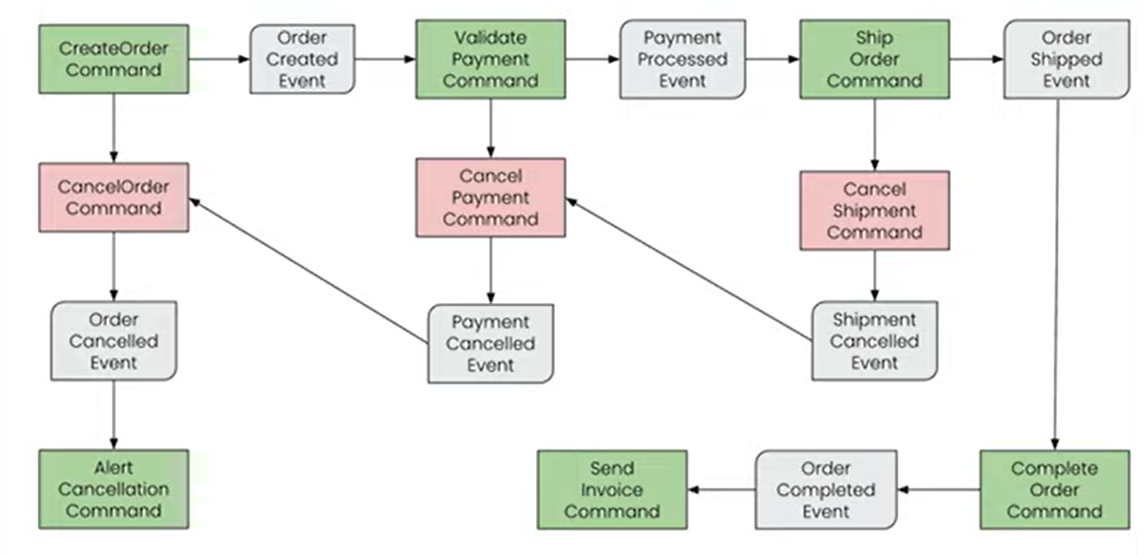


So here we can see that whenever our particular API will be called there will be a **Controller** that controller will handle the API Request that is **POST** for the create order and it will create one particular **Command** **CreateOrderCommand** my **OrderSaga** this is everything about my **OrderSaga**. So, this particular OrderSaga will have one command that is **CreateOrderCommand.** So, once I have the particular command there will be always 2 scenarios right there will be a positive scenario and there will be a negative scenario, success, and failure so we will see both.

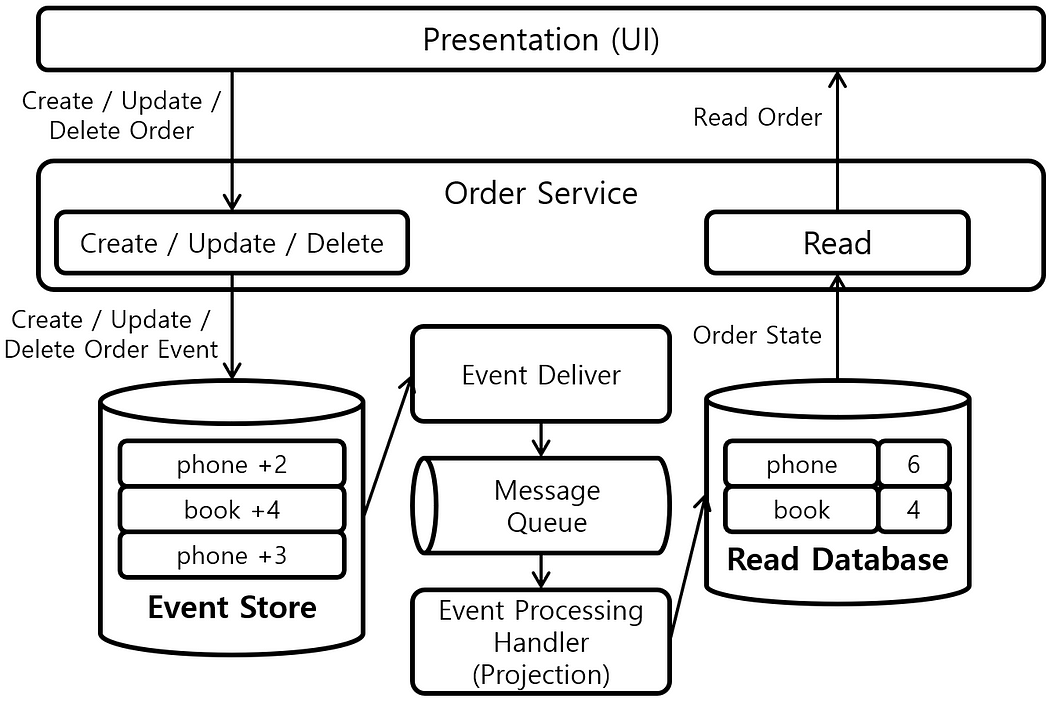
So, when there is a success from that particular command, at that time we will create the **OrderCreatedEvent**. So, for every command we will be creating the event. So, once the particular command is successful, we will create the Event for it and if it’s not successful we will be creating the compensating event for it. So, this is a **CreateOrderCommand** and if that is successful **OrderCreatedEvent** will be created and if it’s not successful we will create the **CancelOrderCommand** over here to cancel the particular order and that particular **CancelOrderCommand** will create the OrderCancelledEvent to cancel the particular order. Now if the scenario is successful and there is no issue in creating the order after that we will create the new command that is the **ValidatePaymentCommand** to validate my payment for whatever the order we have got. And for that if every validation and everything is successful, we will create the event that is called **PaymentProcessedEvent** and if it was a failure scenario, we will create a command that is **CancelPaymentCommand** and this particular **CancelPaymentCommand** will create the event that is called **PaymentCancelledEvent**.

Now this particular **PaymentCancelledEvent** should call the **CancelOrderCommand** as well. Because we need to have the compensating transaction for everything. So, once my payment is cancelled, we need to cancel the order as well. So this particular **PaymentCancelledEvent** will again call the **CreateOrderCommand** that will cancel the order for me. Let’s come to the positive scenario I have create the **PaymentProcessedEvent** when payment validation was successful. After that we will create a new command that is **ShipOrderCommand**. Now my payment is successful now let’s ship that particular product and, in that scenario, as well if its successful we will create a new event that is **OrderShippedEvent** and if it’s not successful we will create the new command that is **CancelShipmentCommand** that again will create an event that is **ShipmentCancelledEvent**. So, there is an issue in my shipment right so that shipment should again cancel my payment as well so here you can see that **ShipmentCancelledEvent** again calling the **CancelPaymentCommand** that will again call the **PaymentCancelledEvent** that will result in the cancellation of the order. So, here you can see that there will be a chain of commands that will be executed.

Now let’s get back to the success scenario. My **ShipOrderCommand** was successful and that resulted in the **OrderShippedEvent** so that particular **OrderShippedEvent** will result in the **CompleteOrderCommand** that means my order will be completed at that particular time my Saga should also be completed. So, here you can see that **CompleteOrderCommand** is successful it will result in the **OrderCompletedEvent** and once the **OrderCompletedEvent** it can send the command like **SendInvoiceCommand** to send the invoice and when the **OrderCancelledEvent** was executed at that time we also need to notify the user like your cancellation was done for your particular order because of this particular issues so, at that time we will also send the **AlertCancellationCommand** to the system to send the particular cancellation to the user. So here you can see that we have the success scenarios, and we have the failure scenarios as well and you can see everything is linked together. So, for every point of time wherever you are we will be creating the compensating transaction for it.



**5. Event Sourcing**



Event sourcing is a pattern used in software design that involves modeling the state of an application as a sequence of events. Instead of storing the current state of the application, the system stores a sequence of events.

How event sourcing works:

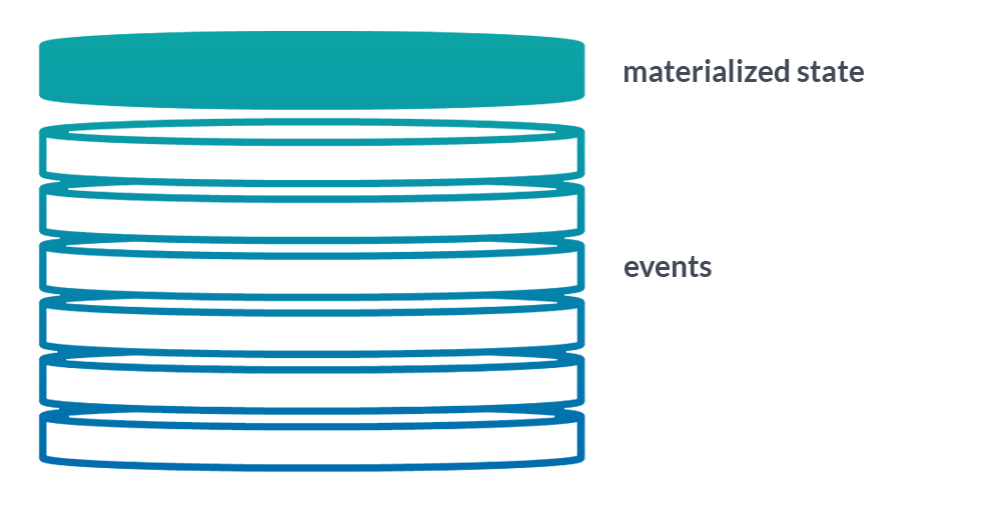
The event sourcing pattern works by storing a sequence of events that describe changes to the state of an application. Each event is stored as a separate record in an event store, which is a database that is optimized for storing large numbers of events.

When a user interacts with the system, the system generates an event that describes the user’s action. For example, if a user places an order, the system would generate an event that describes the order details. The event would be stored in the event store, and the system would update the state of the application based on the event.

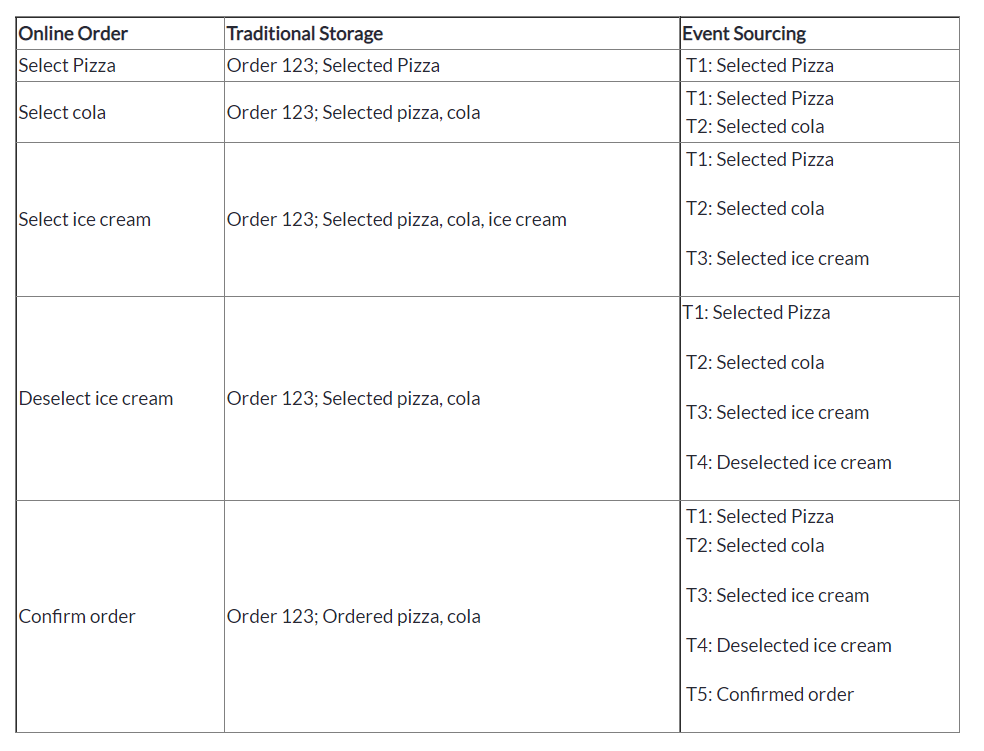
When the state of the application needs to be queried, the system reads all of the events in the event store and applies them in sequence to reconstruct the current state of the application. This process is known as event replay.

In a traditional way of storing an application’s state, we capture the current state and store it in some relational or [NoSQL](https://en.wikipedia.org/wiki/NoSQL) database. While this approach is really straightforward, it does not provide a way of deducting how we got to that current state. Of course, one may argue that we can have a separate model for keeping the history of actions that lead to the current state. Still, besides the additional complexity, these two models could easily go different paths and start being inconsistent (which is basically an [update anomaly](https://www.sqa.org.uk/e-learning/MDBS01CD/page_22.htm)).

Event Sourcing is a way of storing an application’s state through the history of events that have happened in the past. The current state is reconstructed based on the full history of events, where each event represents a change or fact in our application. Events give us a single source of truth about what happened in our application. It is especially beneficial in applications that need to provide a full audit log to external reviewers.



So, AXON Server using this Event Sourcing technique to store all of the particular events. So, whenever we create a particular event from our Command Gateway all the particular events will be store to our Event Store over here.



So, here lots of events is happening. All these particular events like selecting the pizza, selecting the cola then selecting the ice-cream and then de-selecting the ice-cream and then performing the order. all these particular events that are happening are stored inside the **Event Data Store** and all these particular events at the ends will result in a single data that has been read by the **Read Database**. all these particular events executed and results in a single order over here and that single order will be store in a database over here.

Why this particular event-sourcing is so helpful is whenever you have an issue, or you want to re-create the same order you can play the entire this particular event source from start to end to create the same data that you want. You can always go and do select a pizza, select a cola, select an ice-cream and de-select an ice-cream and order it. So, in that way you were able to re-create the same order multiple times for our debugging purposes or you want to create a new database lot of things u can do using the event-sourcing. You can do the snapshots and event Replaying as well.

