5/15/2021

Bipin Kumar

shael.dhn88@gmail.com

**Microservice**

Microservice using CQRS, SAGA, and Axon framework

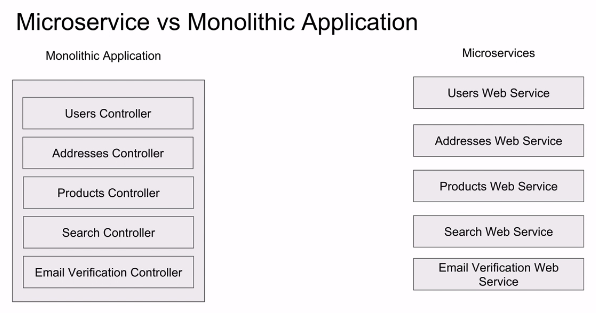
**What is Microservice?**

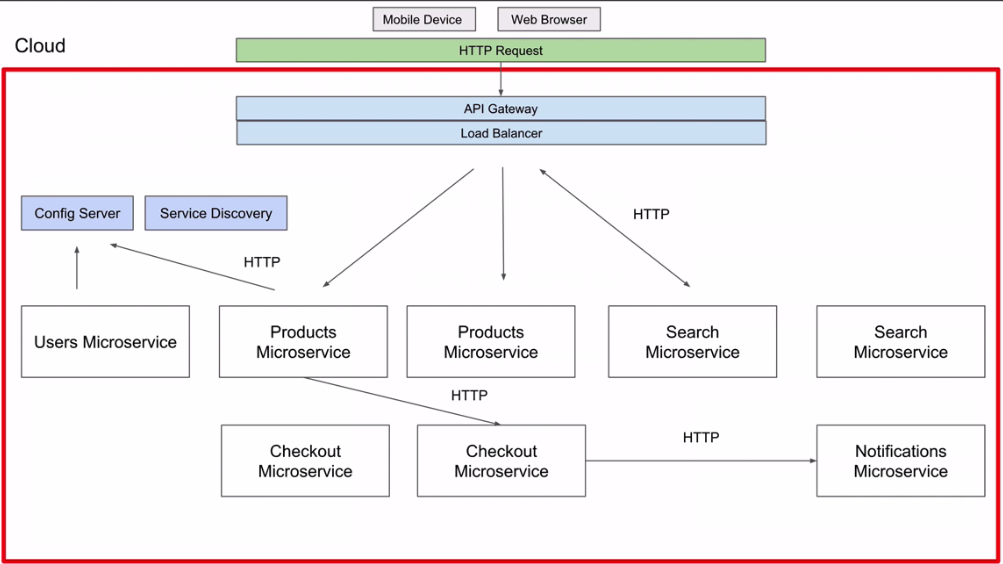
Microservice are a software development technique, or a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services.

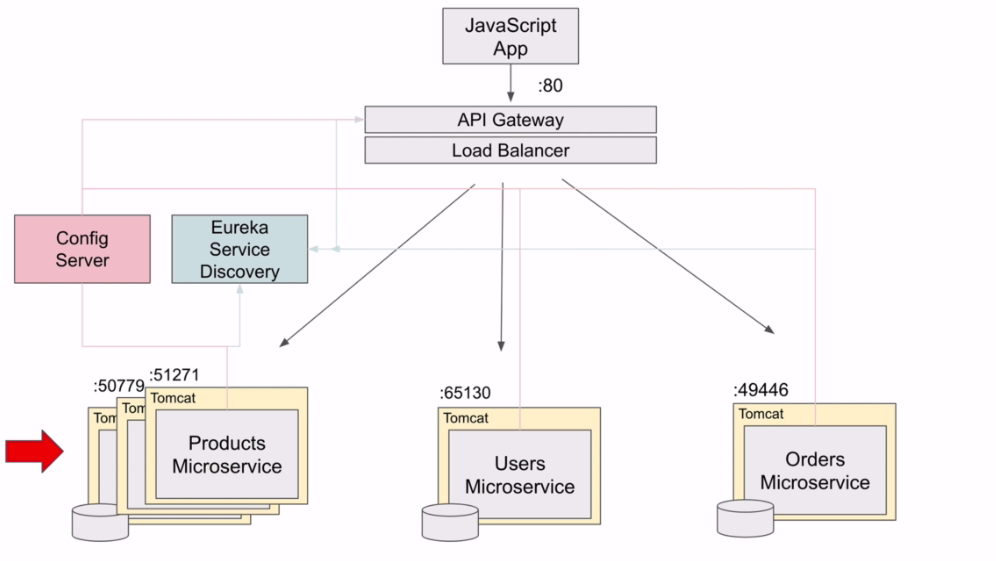
In a microservice architecture, services are fine-grained.

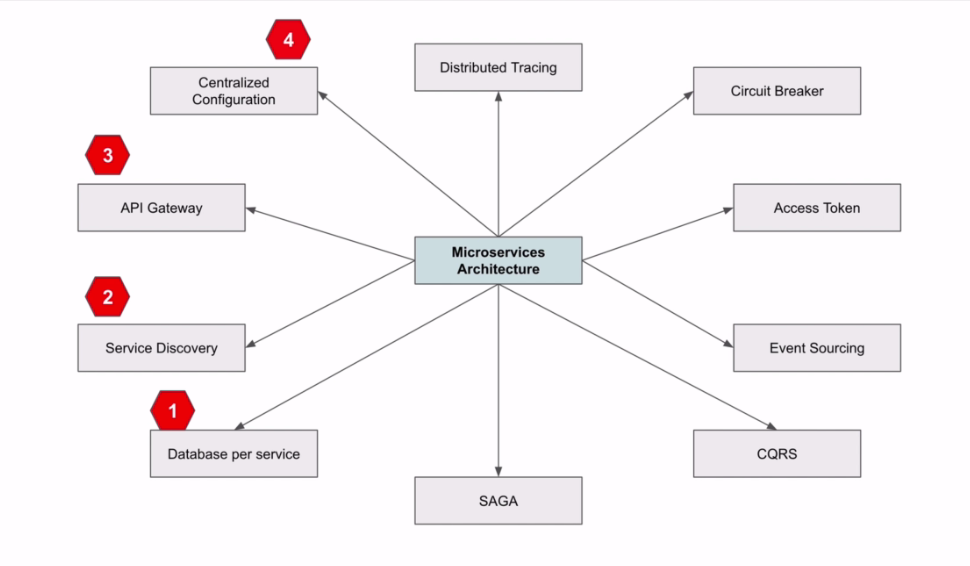
The benefit of decomposing an application into different smaller services is that it improves modularity. This makes the application easier to understand, develop, test and become more resilient to architecture erosion.

It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently.

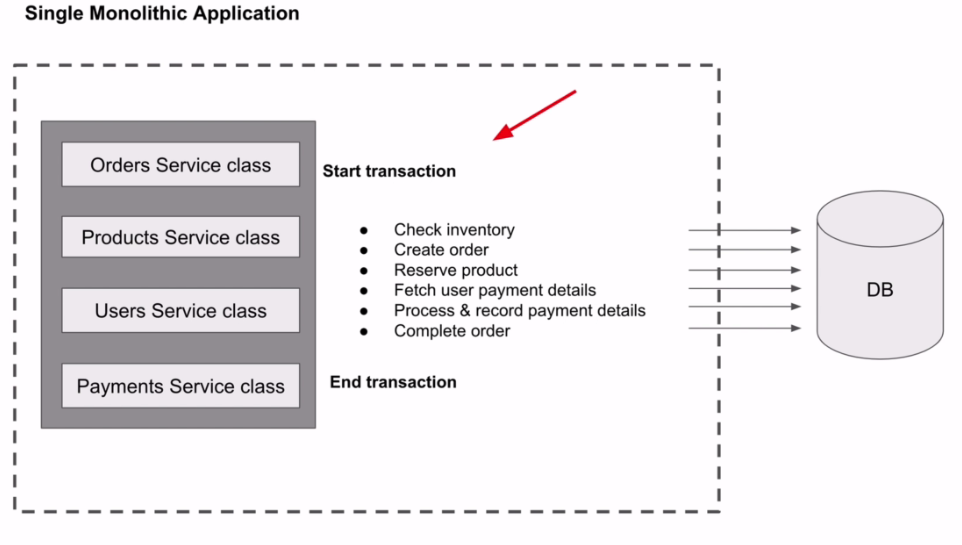


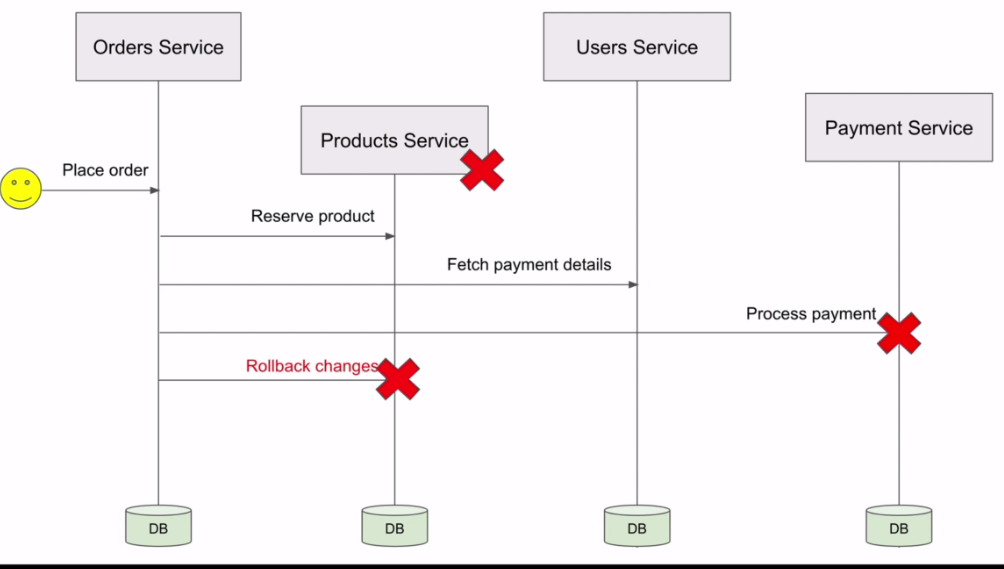






Transaction in Microservices:

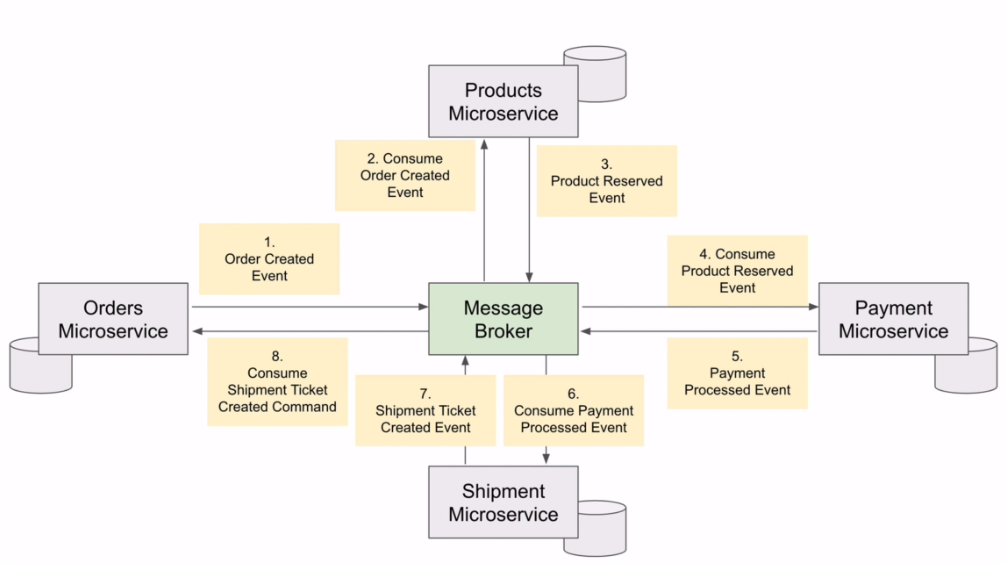




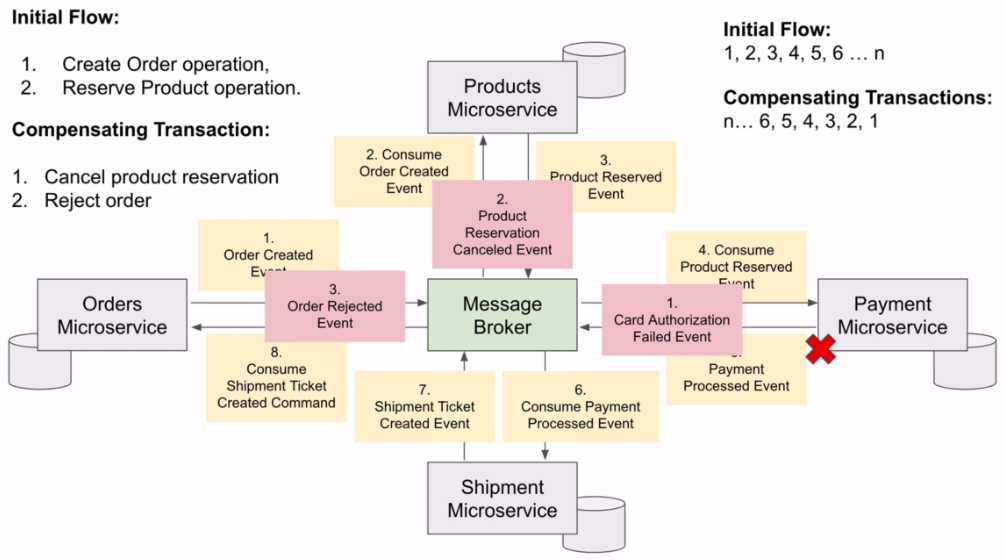
Saga design pattern:

* Choreography saga design pattern:

In choreography-based saga each microservice is responsible for its own transaction and once transaction complete it will raise an event so that another microservice who needs that event notified.

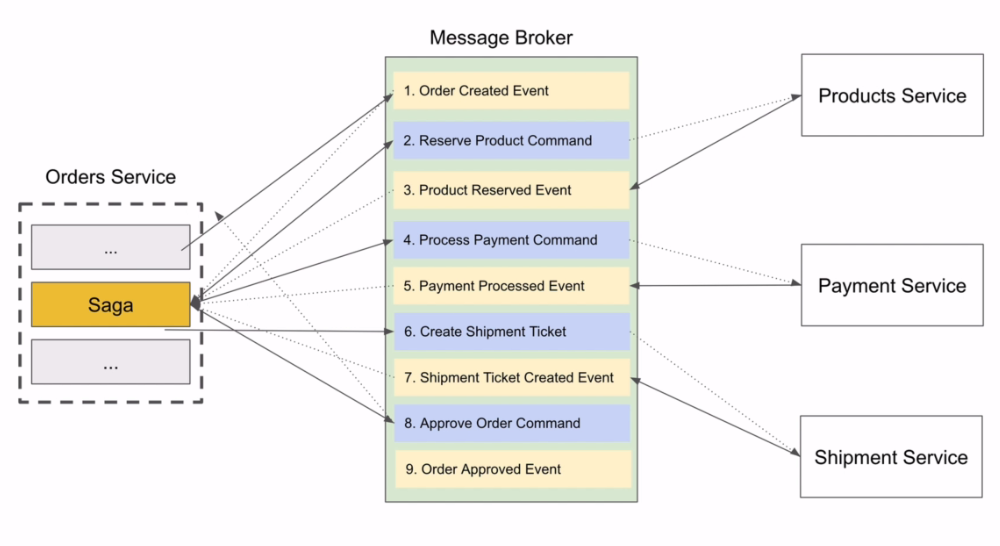


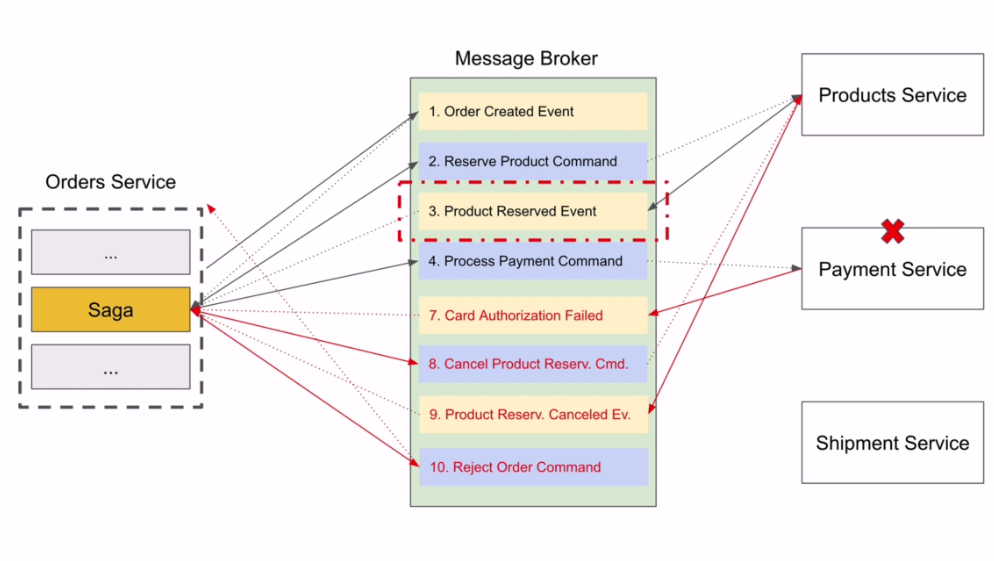
Similarly, if any steps in above diagram failed it will raise an event so that other microservices which already perform transaction will execute compensating transaction or rollback transaction.



* Orchestration-Based Saga

In orchestration-based saga there is central saga which manage all the transaction as shown in below diagram.





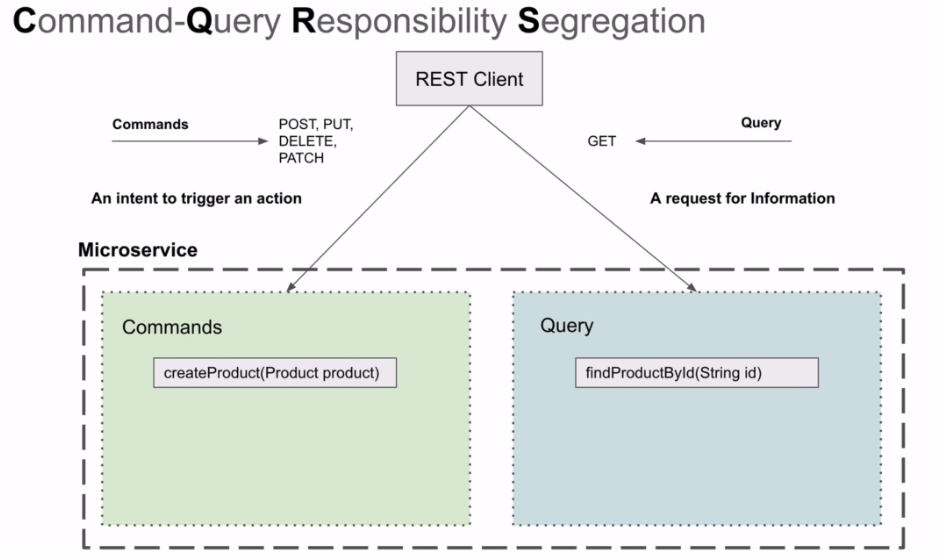
Framework we can use:

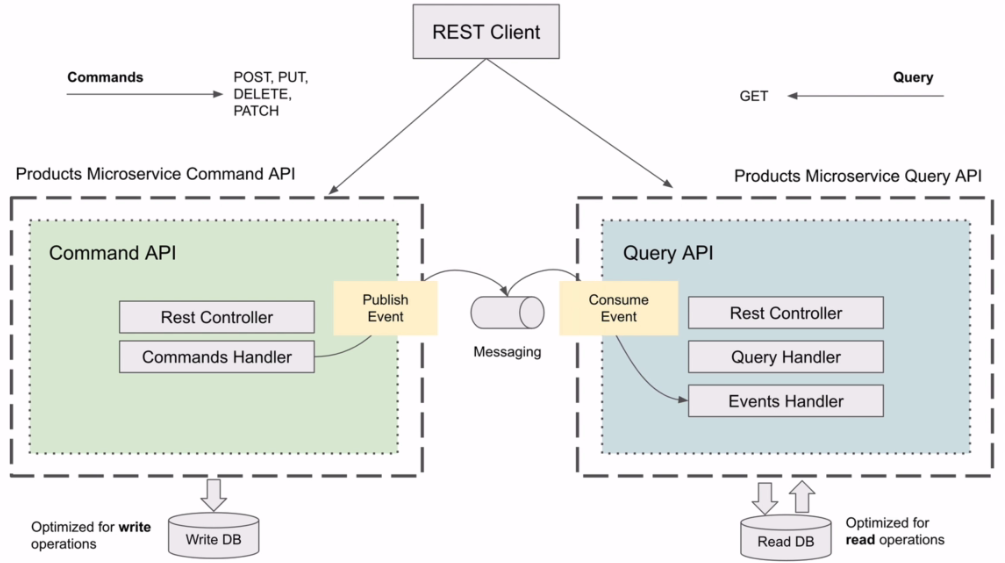
* Eventuate: It support two types of framework.

1. Eventuate Tram: it uses traditional persistence
2. Eventuate local: it uses event sourcing.

* Axon : An open source framework which use DDD and event-driven microservices

CQRS: Command-Query responsibility segregation

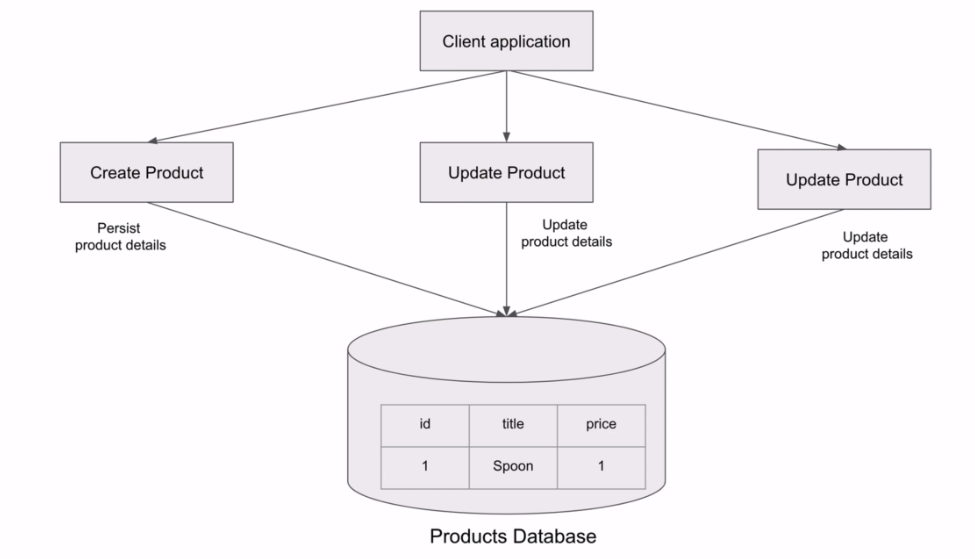


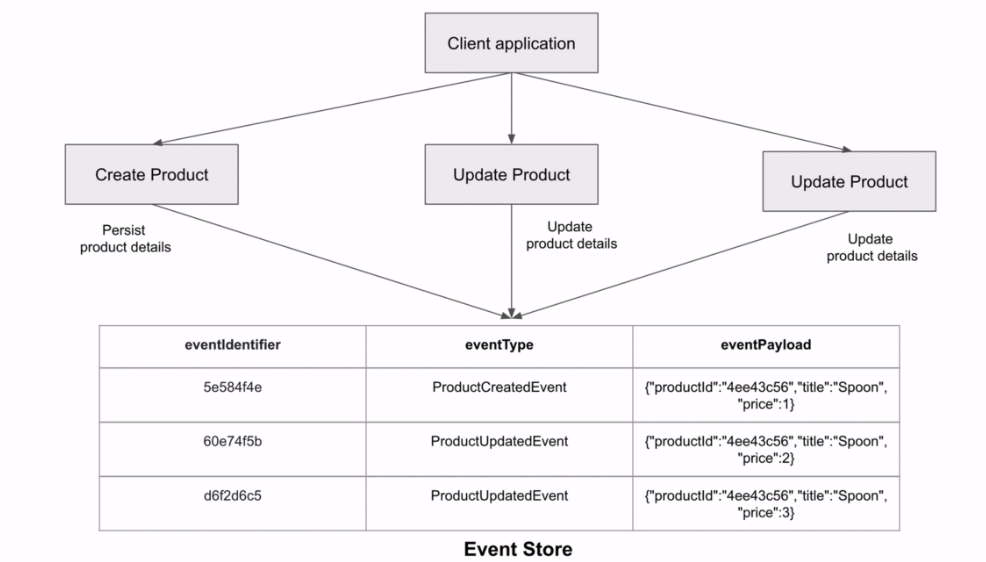


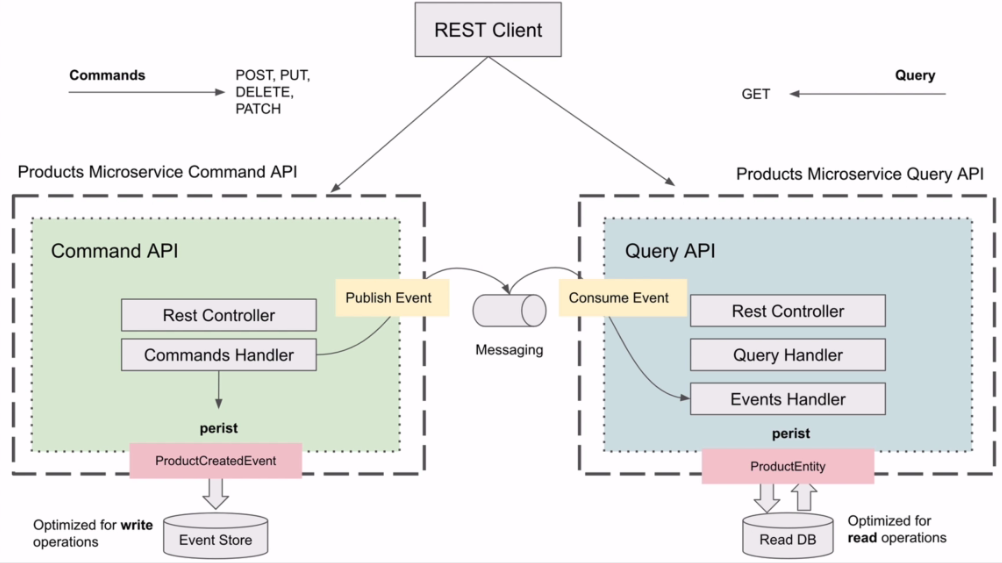
Types of messages:

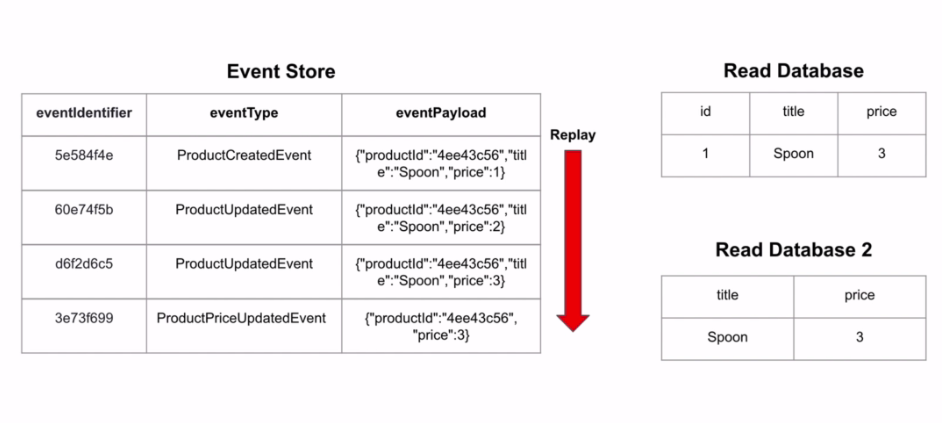
* Command: express the intent to change application state. Like update, delete create
* Query: express the desire to get information. Like getproductbyId
* Event: Represent a notification that something happened.

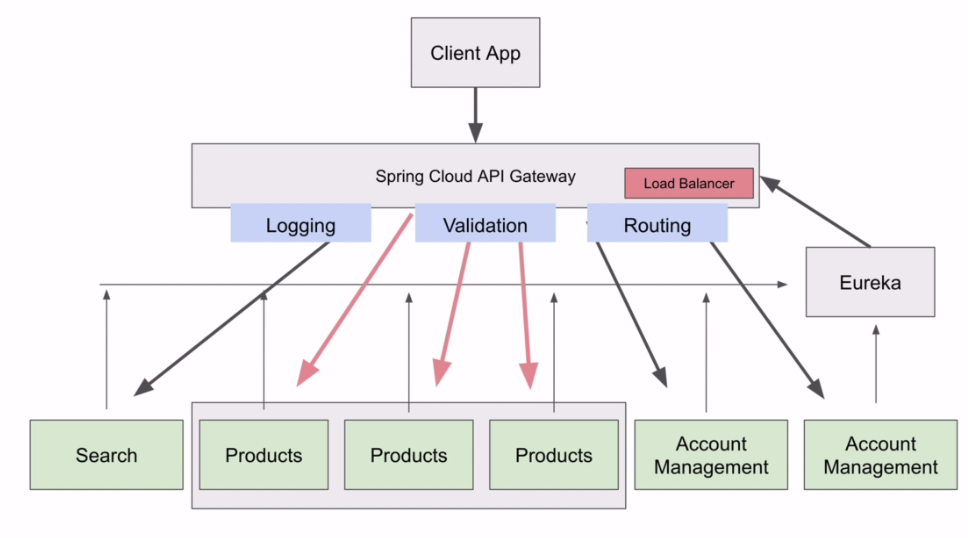
Event sourcing:











Start Spring boot with random port no.:

Spring.port=0 (random choose port no. which is available)

If we use above configuration we can see if you start two or more instance of same service then you will only see last instance started is register to eureka to fix this we need to configure one more property.

Axon Server:

<https://docs.axoniq.io/reference-guide/axon-server/administration/admin-configuration/configuration#configuration-properties>

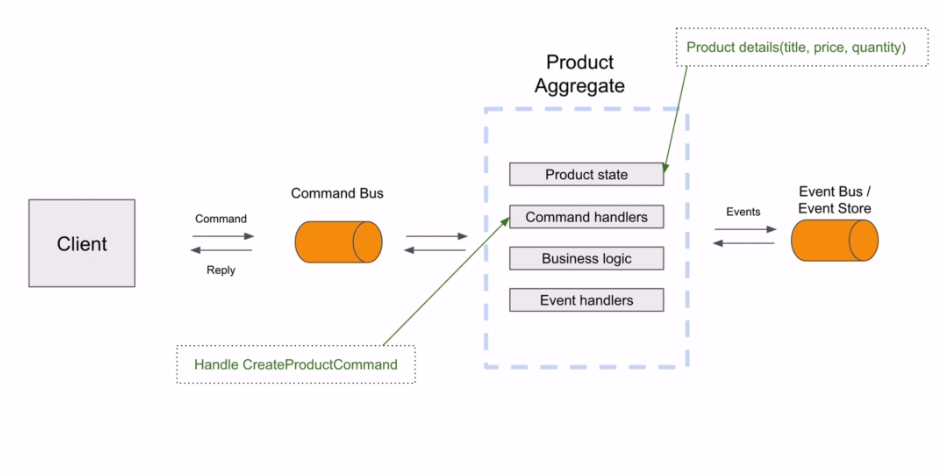
1. Go to the folder where you kept axon server jar
2. Create folder config and create file axonserver.properties
3. Put blow properties in that file and then start axon server

Server.port=8025

axoniq*.*axonserver.name=my Axon server

axoniq*.*axonserver.hostname=localhost

axoniq*.*axonserver.devmode.enabled=true



#### Hibernate Validator

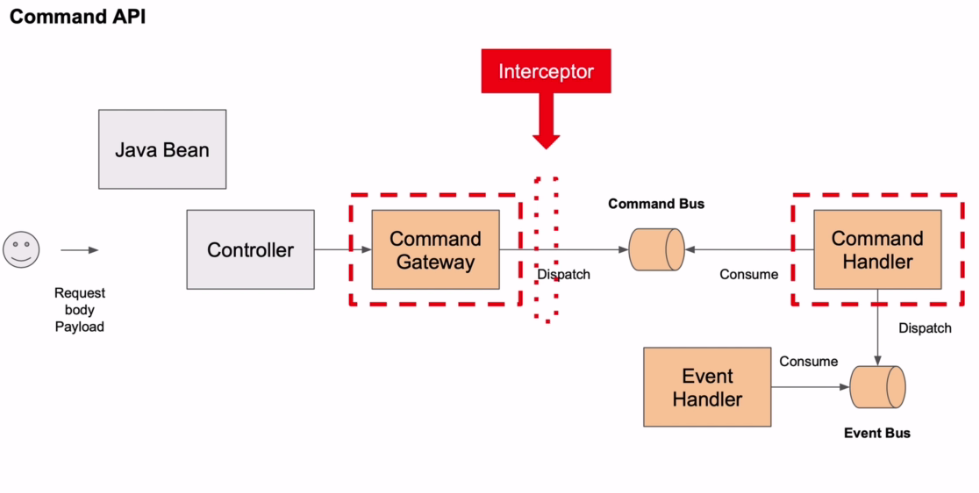
<http://hibernate.org/validator/>

#### Hibernate Validator Documentation

<http://hibernate.org/validator/documentation/>

CommandHandler validation:

To write validation code we can use message dispatcher interceptor. To implement we can MessageDispatchInterceptor



To register message dispatcher, open main class and put below code:

@Autowired

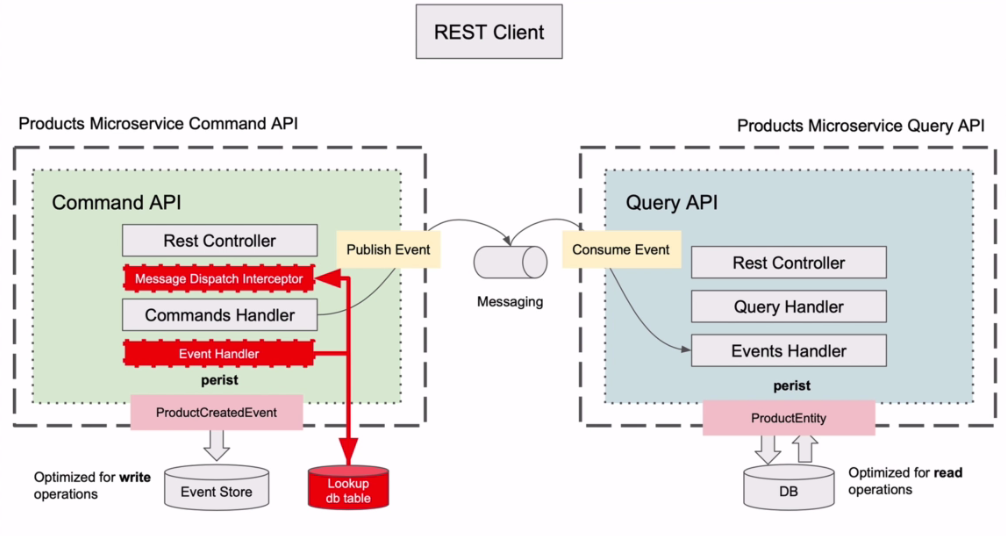
public void registerCommandMessageInterceptor(ApplicationContext context, CommandBus commandBus) {

commandBus.registerDispatchInterceptor(context.getBean(CreateProductCommandInterceptor.class));

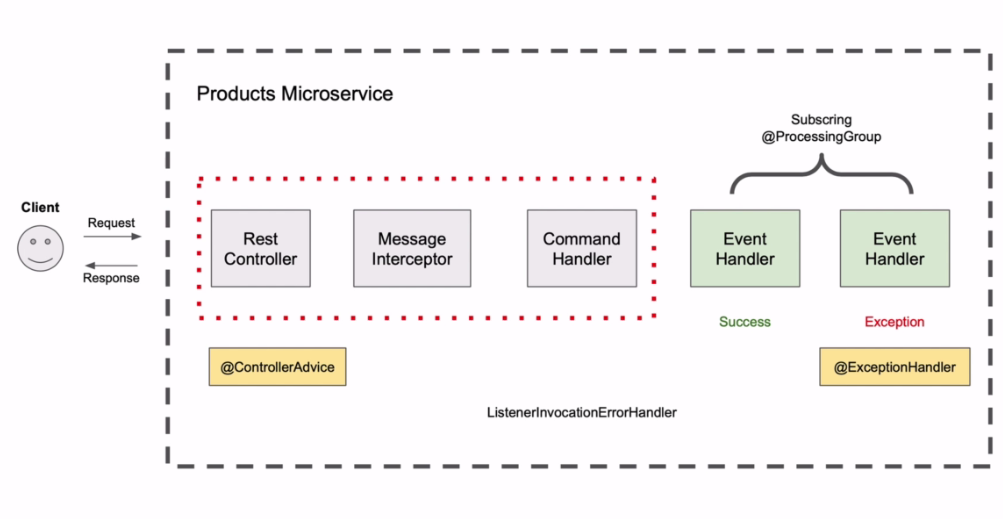
}

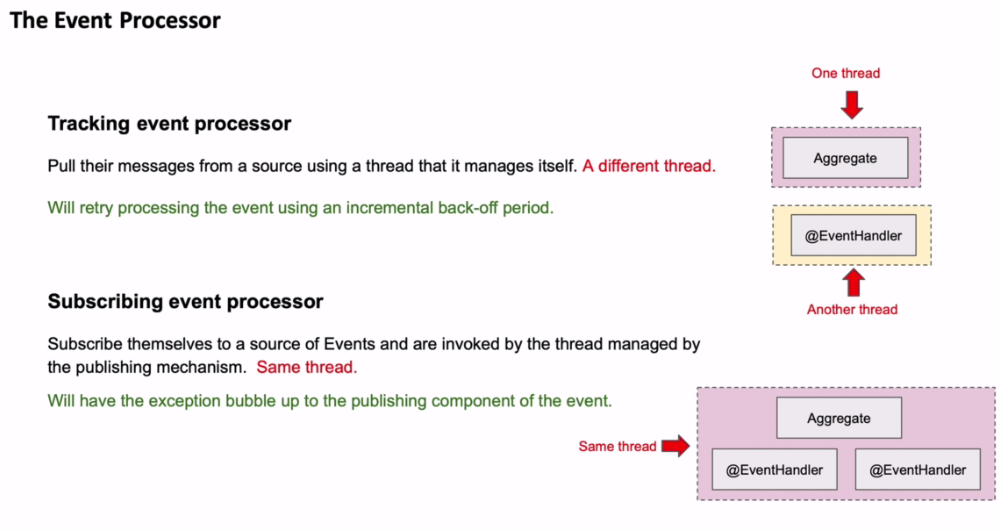
Set Based consistency:

axon.eventhandling.processors.product-group.mode=subscribing



Handle Error and rollback transaction:





Central Exception handler:

To create a central exception handler, create a class and annotate it with @ControllerAdivce and create below type of method to handle specific exception.

@ExceptionHandler(value = { CommandExecutionException.**class** })

**public** ResponseEntity<Object> handleCommandExecutionException(CommandExecutionException ex, WebRequest request) {

ErrorMessage error = **new** ErrorMessage(**new** Date(), ex.getMessage());

**return** **new** ResponseEntity<>(error, **new** HttpHeaders(), HttpStatus.***INTERNAL\_SERVER\_ERROR***);

}

Above example is for handling any exception if happened after calling below method.

AggregateLifecycle.*apply*(event);

You can see if you write any customize code to create exception after above method it is not calling the event method. It is because axon framework not persist the event once this method called.

There may be possibility of having and exception during event handling so for that we can handle that using @ExceptionHandler annotation provided by axon framework.

ListenerInvocationErrorHandler:

1. Create a class implementing above interface.
2. Register that class using below code.

@Autowired

**public** **void** configure(EventProcessingConfigurer config) {

config.registerListenerInvocationErrorHandler("product-group",

conf-> **new** ProductServiceEventErrorHandler());

/\*

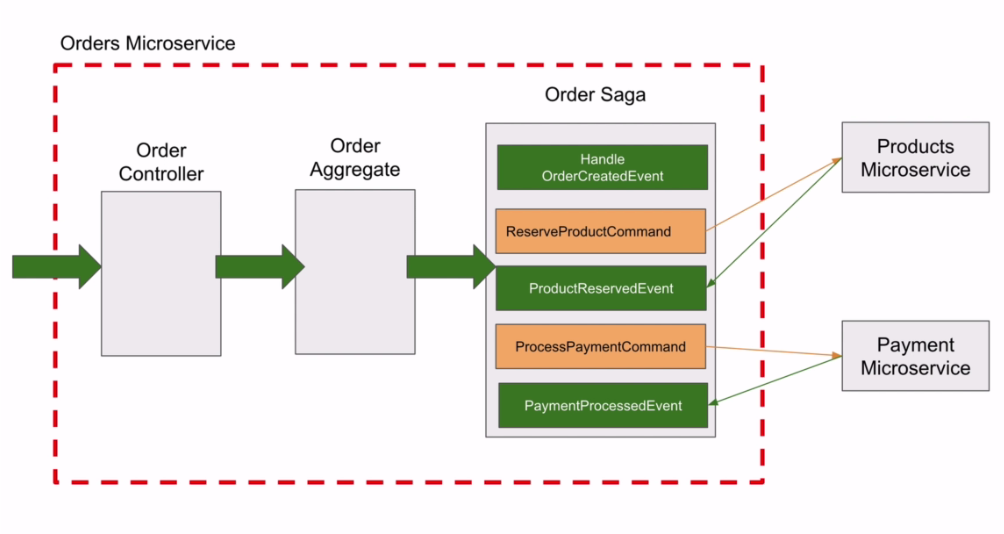
\* config.registerListenerInvocationErrorHandler("product-group", conf->

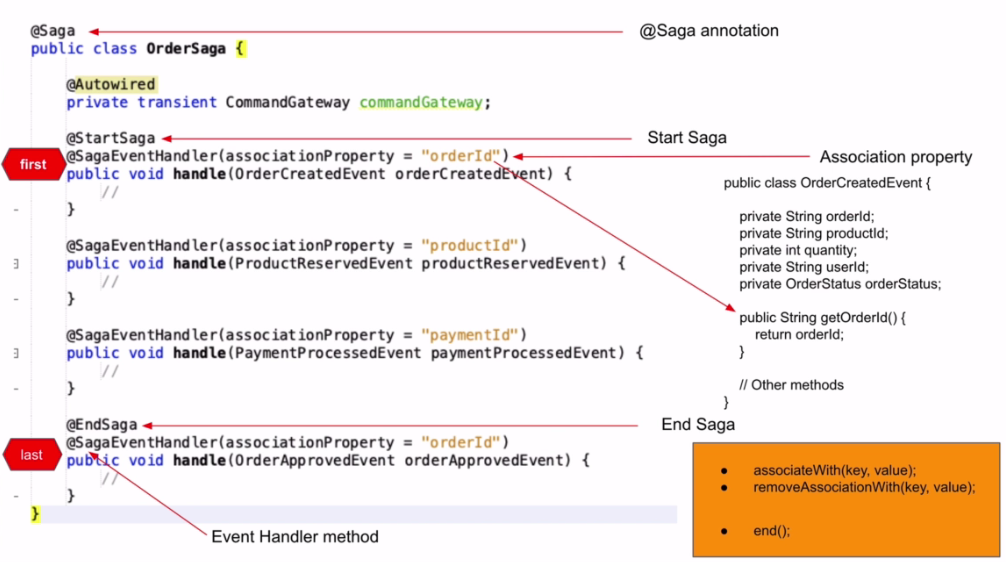
\* PropagatingErrorHandler.instance());

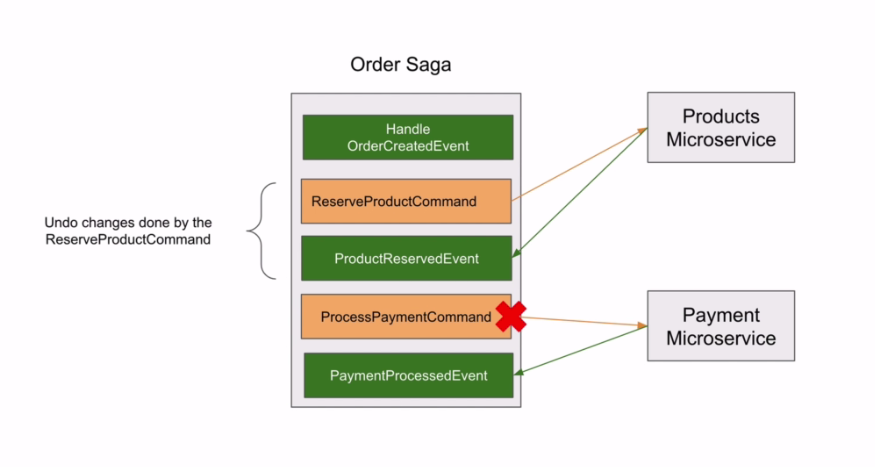
\*/

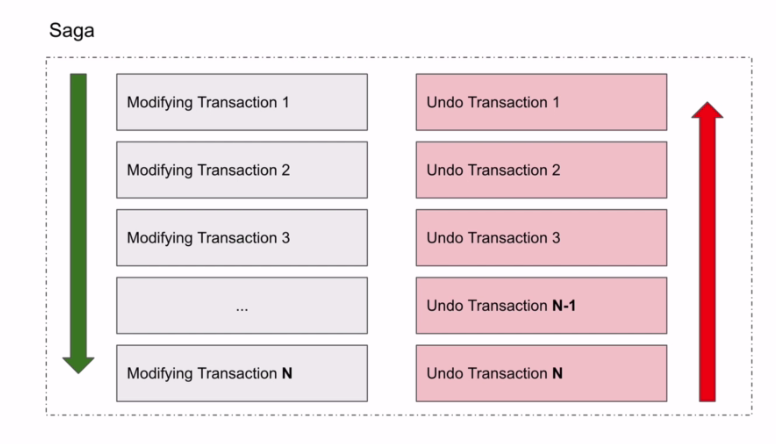
}

Orchestration based SAGA:









Deadlines:

* Deadline is an event that takes place of an absence of any event
* Can be used in saga as well as outside of saga
* Trigger a state change or a command
* Is not “Sourced” means not persisted in DB
* Triggered only one time.

Steps to create Deadlines:

* Create a DeadlineManager.
* SimpleDeadlineManager: It keeps the deadline in memory and don’t persist it .

@Bean

**public** DeadlineManager deadlineManager(Configuration configuration, SpringTransactionManager transactionManager) {

**return** SimpleDeadlineManager.*builder*().scopeAwareProvider(**new** ConfigurationScopeAwareProvider(configuration))

.transactionManager(transactionManager).build();

}

* QuartzDeadlineManager : It uses persistence task to keep deadline

@Bean

**public** DeadlineManager deadlineManager(Scheduler scheduler,AxonConfiguration configuration,

SpringTransactionManager transactionManager, Serializer serializer) {

**return** QuartzDeadlineManager.*builder*()

.scopeAwareProvider(**new** ConfigurationScopeAwareProvider(configuration))

.serializer(serializer)

.transactionManager(transactionManager)

.scheduler(scheduler).build();

}

* Schedule a new deadline

@Autowired

**private** **transient** DeadlineManager deadlineManager;

scheduleId = deadlineManager.schedule(Duration.*of*(120, ChronoUnit.***SECONDS***), PAYMENT\_PROCESSING\_TIMEOUT\_DEADLINE,

event);

* Handle Deadline:

//In case deadline not

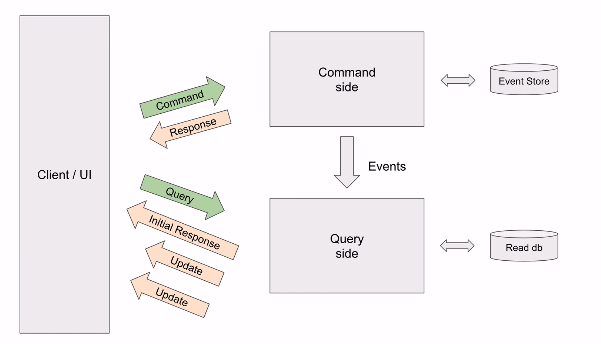
@DeadlineHandler(deadlineName = PAYMENT\_PROCESSING\_TIMEOUT\_DEADLINE)

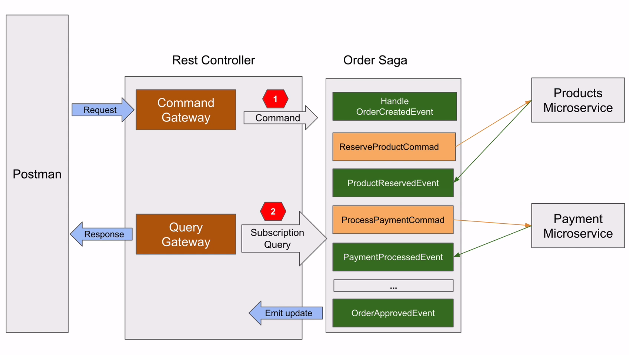
**public** **void** handlePaymentDeadline(ProductReservedEvent productReservedEvent) {

***// send compensation transaction***

}

Subscription Query:





1. Create a query gateway in controller class and create subscription object.

**private** QueryGateway queryGateway;

SubscriptionQueryResult<OrderSummary, OrderSummary> queryResult = queryGateway.subscriptionQuery(

**new** FindOrderQuery(orderId), ResponseTypes.*instanceOf*(OrderSummary.**class**),

ResponseTypes.*instanceOf*(OrderSummary.**class**));

**try** {

commandGateway.sendAndWait(createOrderCommand);

**return** queryResult.updates().blockFirst();

} **finally** {

queryResult.close();

}

1. Create dto object for quey i.e. ordersummary.
2. Create query handler class and create a method which is annotated with QueryHandler annotation write the logic to fetch from DB and convert it to order summary object.
3. Now go to ordersaga class and write below code

@Autowired

**private** **transient** QueryUpdateEmitter queryUpdateEmitter;

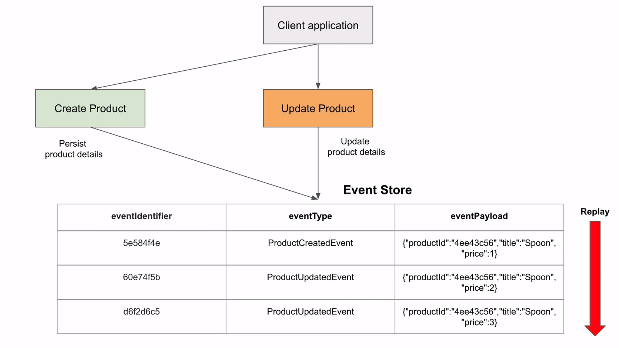
Write below code where for emit.

queryUpdateEmitter.emit(FindOrderQuery.**class**, query -> **true**,

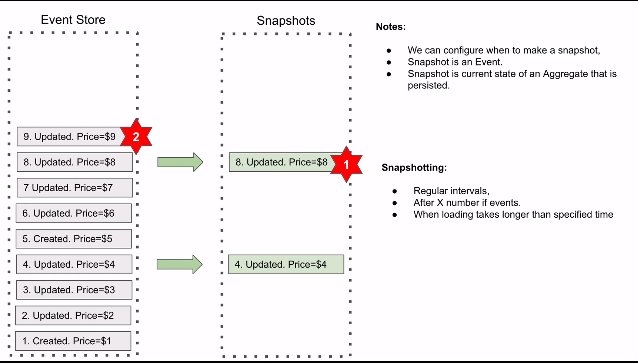
**new** OrderSummary(event.getOrderId(), event.getOrderStatus(), "Order Successful"));

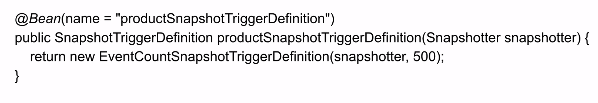
Snapshots:

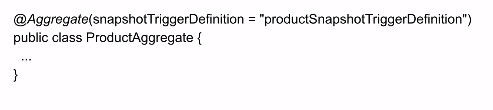
It is optimization feature introduce by axon framework, that allow us to create snapshots, it helps us to speed up the process of loading events from the event store.



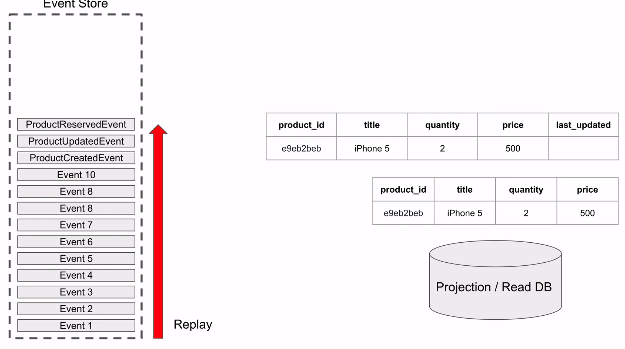
Here every event is stored in event store, when we give a command to update product it will create a product aggregate and then fetch all the events and reply it to get current state of product.

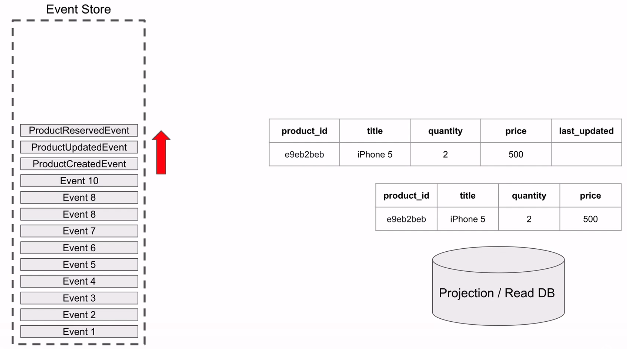






Replaying Event:





* Event Replays is supported only with the Tracking event processor
* Use @DissallowReplay to exclude @EventHandler methods during event replay,
* Stop the current tracking event processor.
* @ResetHandler: mark the annotation to method which you want to do before replay.

@ResetHandler

**public** **void** reset() {

repo.deleteAll();

}

@RestController

@RequestMapping("/api/v1/management")

**public** **class** EventsReplayController {

@Autowired

**private** EventProcessingConfiguration eventProcessingConfiguration;

@PostMapping("/eventProcessor/{processorName}/reset")

**public** ResponseEntity<String> replayEvents(@PathVariable String processorName) {

Optional<TrackingEventProcessor> trackingEventProcessor =

eventProcessingConfiguration.eventProcessor(processorName, TrackingEventProcessor.**class**);

**if**(trackingEventProcessor.isPresent()) {

TrackingEventProcessor eventProcessor = trackingEventProcessor.get();

eventProcessor.shutDown();

eventProcessor.resetTokens();

eventProcessor.start();

**return** ResponseEntity.*ok*()

.body(String.*format*("The event processor with a name [%s] has been reset",

processorName));

} **else** {

**return** ResponseEntity.*badRequest*()

.body(String.*format*("The event processor with a name [%s] is not a tracking event processor."

+ " Only Tracking event processor is supported",processorName));

}

}

}