

Microservices



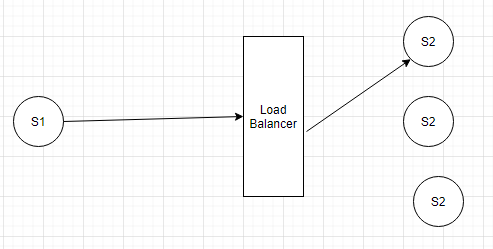
**Microservices**

Microservice can have many definitions.

* It is architectural style or an approach to developing a single application as a suit of small services, each running in its own process and communicating with light weight mechanisms, often HTTP resource API.
* Each service is independently deployable by fully automated deployment tools.
* Each can be written in different programming language. And different persistence can be used. Where as in monolith we have to stick on same language.
* If we compare microservice with monolith application it is easier to understand and developer can be more productive.
* Here we can accomplish parallel development. Where as in Monolith application it may create some issue.
* It can be easily scaled in terms of resources. Where as it is hard to scaled Monolith application.
* Each microservice can be deployable independently, it also improves fault isolation and conflicting resource can be resolved.

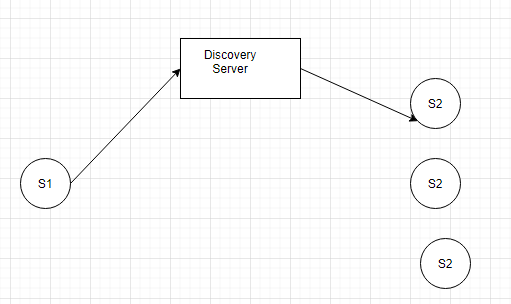
**Challenges and Solution of Microservices:**

1. In microservice generally one service calls another service take as example below.

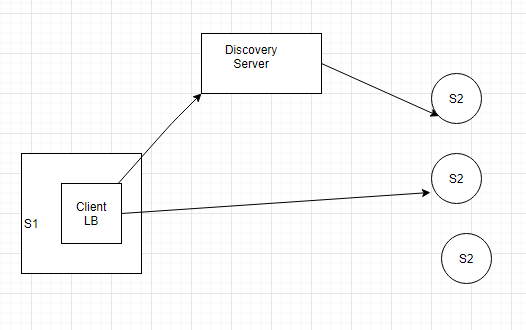


In above example we have two microservice i.e. S1 and S2, S1 is making a call to S2 and based on LB logic instance of S2 is assigned to that call. Here we can see we are having two remote calls. S1 to load balancer and then LB to S2. So, it will affect performance.

Now to over come this issue what microservice architect do that they have introduced another component call discovery server as shown in below diagram.



Discovery server will keep track of all the instance of S2. So, when call coming from S1 it will assign one instance of S2 to serve that request. But still our above problem is not resolve. So now what they do they have introduced another concept called client-side load balancing as shown in below diagram.



Here client side LB download all the registry of S2 instance from of discovery server and when S1 calls S2 based on some logic it will assign S2 instance to full fill the request. But here is another problem is that suppose if after sometime an instance of S2 is down then how client side LB know that S2 instance is down. Also, to download all the registry from discovery server it has to make a call to discovery server.

So, what Microservice architect has done during startup of S1 they have assigned one thread which actually get all the registry from discovery server and also after some seconds based on configuration it continuously synching the registry from discovery server. Similarly, Discovery server also call health check status of each S2 instance after some time interval and based on result it updates its registry and status.

For this purpose, Netflix have created Eureka (Discovery server) and Ribbon (Client side LB). what Netflix developer saw 30 secs is good time to synch and update registry as well as update the registry in Client-side LB.

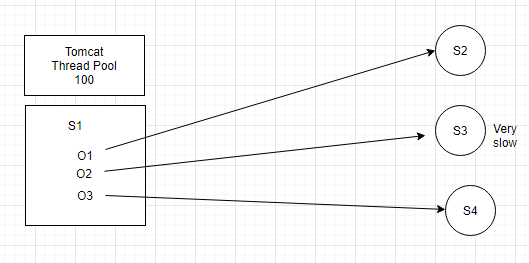
Now suppose during that 30 sec one of instance of S2 went down. Every instance of S2 send heart beat on regular interval (30 sec). Now suppose one of heart beat is missing then what will Eureka do. Should it remove it from registry. No, we can configure that if 3 or more heart beat consecutively missing then only Eureka will remove this server from registry.

Now suppose during this time S1 call that down instance of S2. It got fail so we can configure retry logic, (Ribbon is providing this retry logic). But what should we do if all retry is over then in that case we can implement some fallback logic.

Also take one example if S1 calling one of S2 instance and that instance is busy or taking some time to response. In that case, may S1 can assign weight to that instance and similarly to other instance of S2 and based on this it may use weight round robin to call S2 instance.

1. Bulk-head issue.

Let take an example we have following server S1, S2, S3, S4 etc. And S1 have operation O1, O2 and O3. And S3 service is very slow. We may deploy those service to tomcat or other server and it has some thread pool. Suppose S1 is deployed to tomcat server and its thread pool size is 100. So, what will happen if O2 calls S3 service, due to slowness it might be waiting for response. Similarly, there might be chance that all 100 threads are calling to S3 and due to slowness is waiting for response and our application consume all the thread pool and no thread is available for other operation.

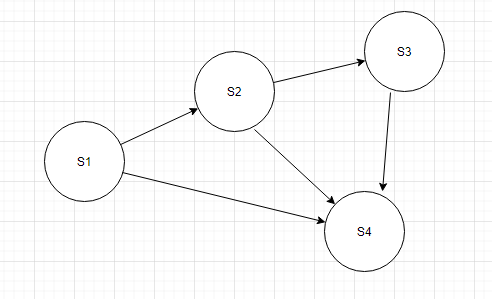


Now take example of bank having toll free no. with 100 limits. Suppose it has many departments like saving account, loan, credit card etc. When caller call the toll, free no. based on it’s choice call connected to different department. Suppose Loan department taking time to describe loan feature and if we don’t have any mechanism then all the 100 limits get used. So, what we can do we can assign some limit to each department like SA can have 30 call max, loan can have 40 calls max and CC can have 30 calls max. So, what will happen in this case if more than 40 callers try to connect to loan department either they will wait or call get disconnected after sometime automatically. By this here we can utilize limits. Means 60 call will be available for another department.

Similarly, in above case of microservice we can limit pool size specific to each service. This pattern is call Bulk head pattern. Means issue in one service could not make entire system down. This concept came from ship companies. We don’t need to implement bulk head pattern Netflix provide one more library called Hystrix which implement bulk head pattern. It also used in circuit breaker pattern.

1. Circuit breaker pattern.

Look below diagram, S1 calling S2, S2 calling S3 and S3 calling S4, S2 calling S4 and S1 calling S4. By the looks of this it looks like a circuit.

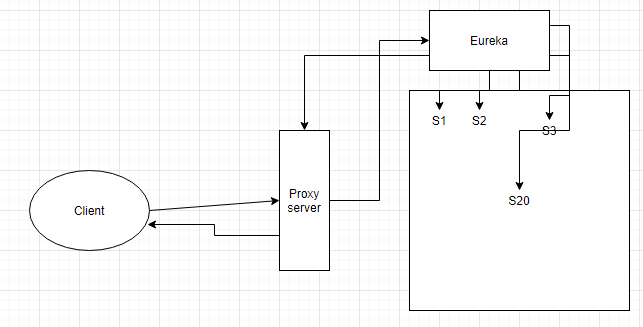


Here suppose due to some reason Service S4 fails, what will happen then after sometime S3 fails then S2 and then complete services.

One thing we can do we can write some fallback logic which will execute once S4 is down. Now suppose S4 is down and S3 is calling S4 1,2… 20 times it got failure and it execute fallback logic. Suppose if we have mechanism which will already identify that S4 is down without wasting time calling S4 it should execute fallback logic. In circuit breaker there are lots of logic which decide whether a circuit is open or close based. E.g. success count, failure count, time out, bucket with data all SC, FC, TC of 100sec etc.

1. Suppose we have 20 services like S1, S2 … S20 all register to Eureka. So, if internal call happens between these services then it will go to eureka and found the service registry and call those service.

Now suppose if any external client wants to use one of your service can you give it all the registry present in Eureka. So instead of giving all the registry present in Eureka we will provide a proxy server. So now client will call proxy server and proxy server will go to eureka get the registry and based on service call return data. Zuul is one of the libraries which is used as API gateway. It also implements hystrix. Suppose I want to secure my all service we will implement at zuul, similarly we can also configure process like pre-processing or post processing of any request. Also, we can write routing logic also if client wants to use only some of the services.



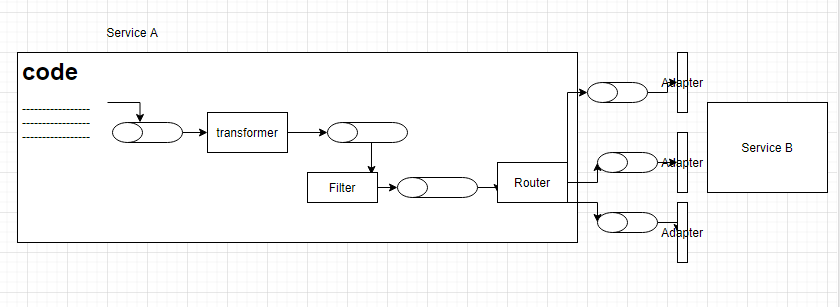
1. In our microservice there can be many services, so ideas are keeping all the configuration inside the central repo. And we should have config server. So, whenever a service starts then it calls config server and it will get from central repo. For this we can use spring cloud config server
2. Message driven microservice using spring cloud stream:

Spring cloud stream build on top of spring integration.

Suppose we have two application A and B. Service B uses Soap based webservices and A consume B service using wsdl file. If we are using Soap based service, till then contract i.e. wsdl is same then we don’t have any impact like B changes it language from java to python etc. We can say application A is loosely coupled with B. But what about logically coupling, logically coupling means application A don’t know whose is application B and vice versa but still they can talk to each other. But is case of soap implementation there is no logically coupling.

Take above example, B is exposing Soap service. Now what A will do, it will create general message like header and body and it has in memory queue in which it will push message. Now suppose we have something called Adapter. Which will take that message from in memory queue and convert it to Soap envelop and call the service exposed by B. In this case A will don’t know about service B. now suppose latter on service stop support for soap and converted it to REST service what we need to do we only need to change configuration from soap to REST. And here everything is done through configuration.

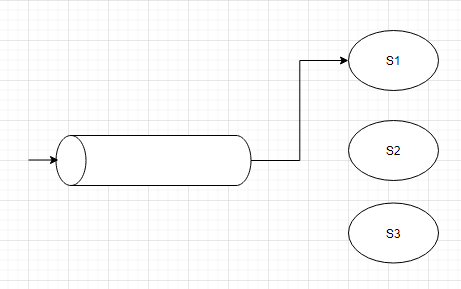
Now take example, we are pushing data as in json format in in-memory queue but service uses xml format so what we can do we put something called transformer which convert data. Now suppose we want to validate data based on some condition then we can also configure filter similarly, we can also configure router which based on some condition put data into another queue and on those queues, we can configure adapter which will call the service.



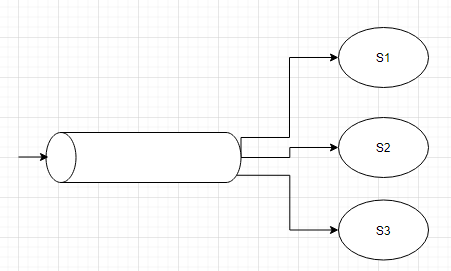
Any infrastructure which let us do above thing called as BUS framework or ESB e.g. MULE, apache camel, spring integration. Here those in-memory queue is called as channel.

Basic of RabbitMq:

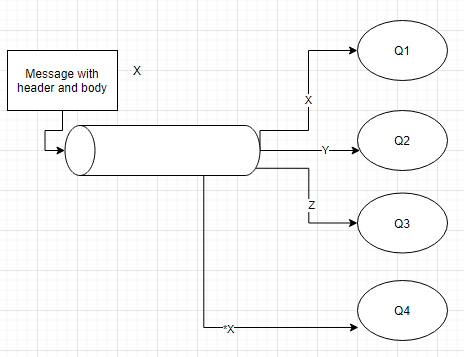
In JMS we have two types of destination, queue and topic. Whenever a producer produce message then receiver receive message. Suppose we have a queue and three subscribers as shown is below diagram in which when a producer produces a message then one of consumer consume this message this is called point-to-point subscription.



Similarly, we have topic, in which when producer sent a message it will be subscribe by multiple subscriber and this pattern is called as pub-sub i.e. publishing and subscribing.



RabbitMQ uses AMQP protocols i.e. Advanced Message Queuing Protocol. So, if we are using AMQP we use different terminology where destination is called exchange. whenever sender wants to send any message we need to create AMQP message and publish it to exchange and it is bounded with multiple queues. Whenever we are bounding queue to exchange we need to define routing key. Now take below example we have exchange which have four queue Q1, Q2, Q3 and Q4 having routing x, y, z and \*x (wild card). So, whenever sender create AMQP message it will be sent with routing header which is placed to exchange and based on routing header it will copied to that queue. In below case message will copied to Q1 and Q2 and it is called topic exchange.



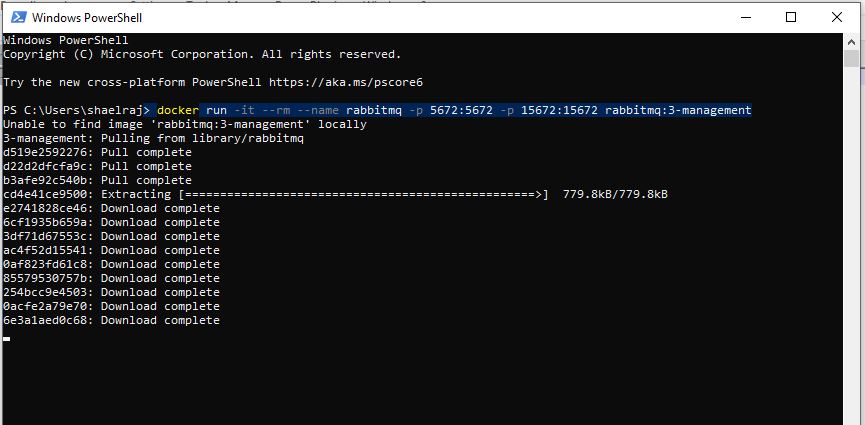
Setup RabbitMQ:

I have used docker for installing RabbitMQ.

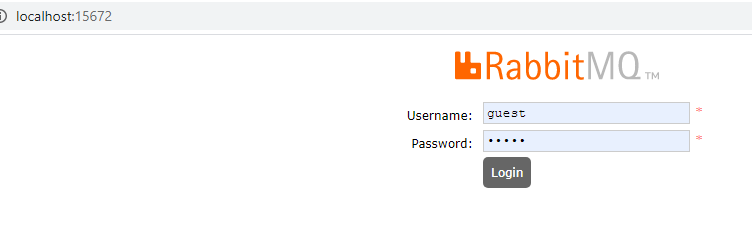
1. Docker should be installed. Run docker and go to power shell or cmd and run below command.

docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management

here it will download image from docker server if image not available locally and installed it to your local machine then it will start your rabbitmq on that docker image. Here -it is used to allow us to provide input/output to the container. similarly -p is used for port mapping because this rabbitmq installed and running in your docker container and it will don’t know when you try to access <http://localhost:15672/> that’s why we used port mapping in docker command so that when a request coming to above URL it will redirected to docker image.



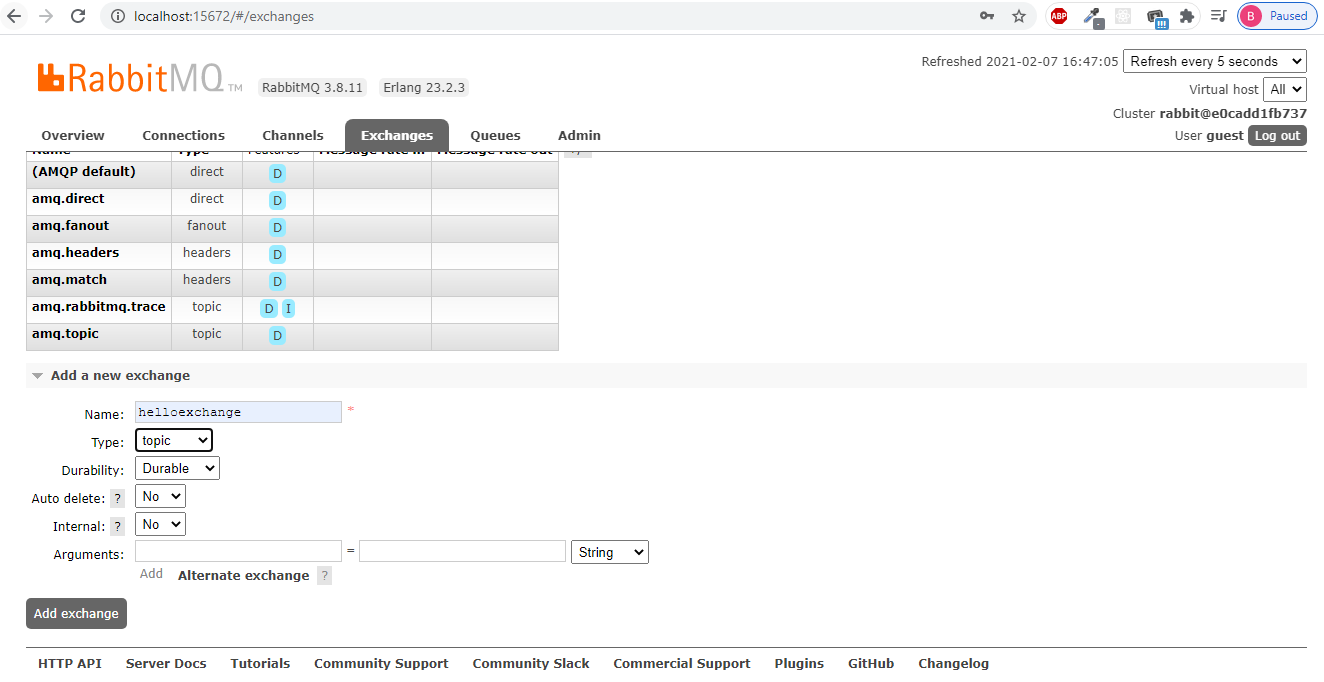
1. Open URL <http://localhost:15672/>



Username: guest

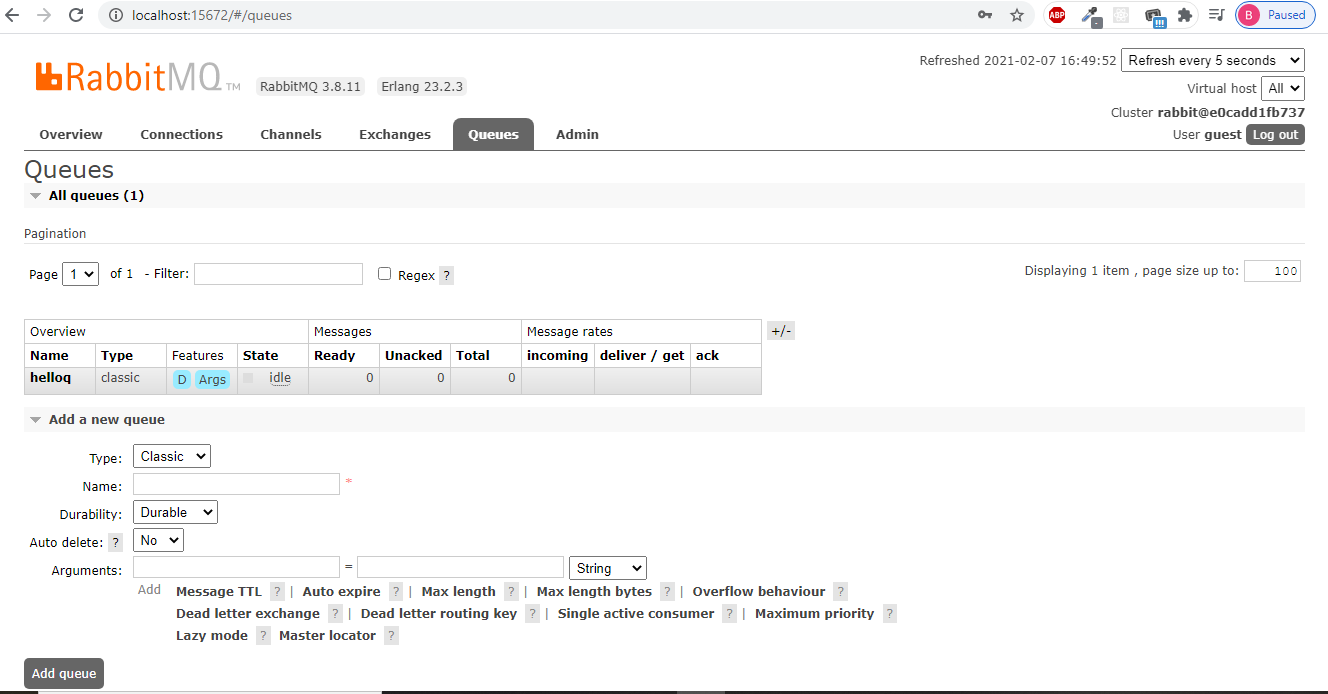
Password: guest

1. Click on exchanges tab

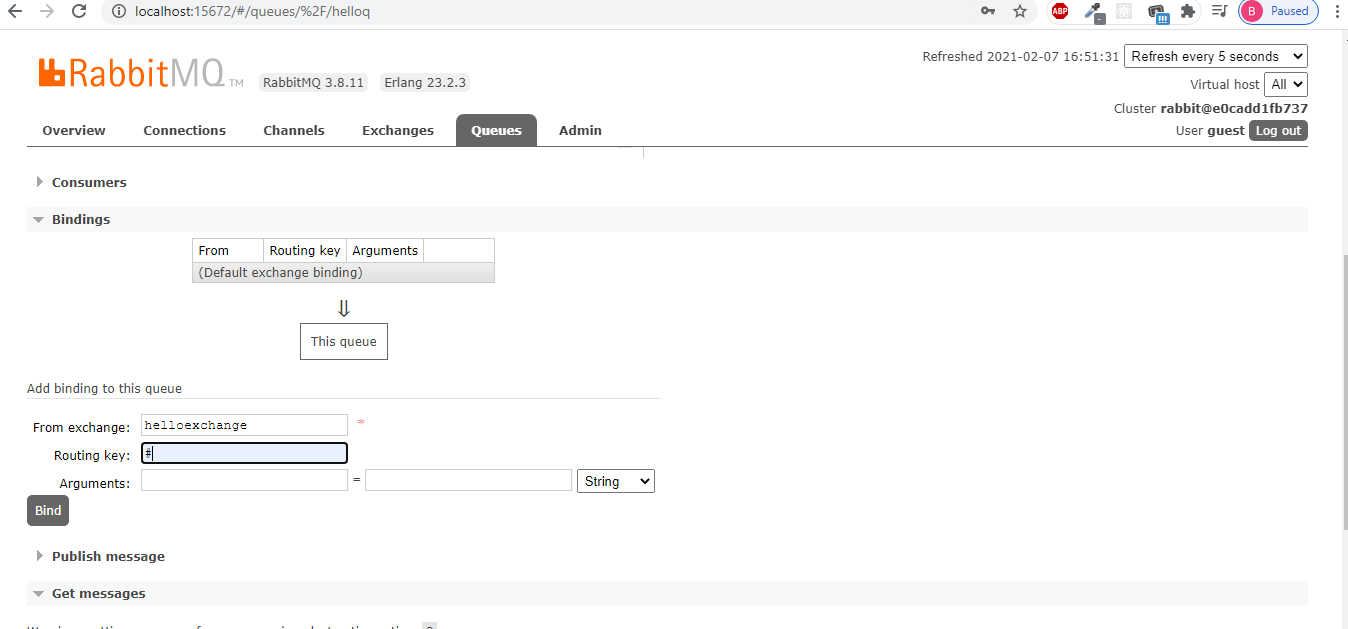


Now click on add a new exchange 🡪 type helloexchange select topic in type combo and click on add exchange button.

1. Now click on queues 🡪 click on add a new queue, given helloq in name. and click on add queue button.



1. Now click on newly created queue and click on binding and enter # in routing



1. Project setup for Producer and consumer product.

Create a spring boot maven project **mdm-producer** and add below dependency.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.3.8.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<groupId>com.javamonks</groupId>

<artifactId>mdm-producer</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>mdm-producer</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>1.8</java.version>

<spring-cloud.version>Hoxton.SR9</spring-cloud.version>

<spring.fox.version>3.0.0</spring.fox.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**<dependency>**

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

**<artifactId>spring-cloud-starter-stream-rabbit</artifactId>**

**</dependency>**

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-boot-starter</artifactId>

<version>${spring.fox.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

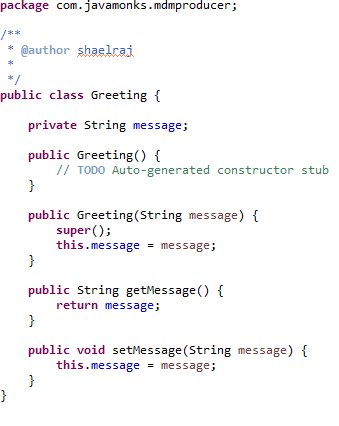
</plugin>

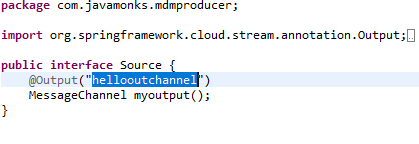
</plugins>

</build>

</project>

In above pom.xml highlighted dependency we have added for RabbitMQ if you want to use for Kafka you can use Kafka dependency.

In our example we can see simple use case like we have greeting message when we call a rest service it will push a message to RabbitMQ. For this we need to create a Greeting class which has only one attribute called message. Similarly, we need to create **source** interface which will define our channel as shown in below screenshot.

In above code we created a channel and here we want our message to go out that’s why we annotated it with @Output and given a channel name.

Now we have to configure this hellooutchannel to exachange for this we need to make configuration in application.properties or application.yml file.

Below is the example of configuration

spring:

cloud:

stream:

bindings:

hellooutchannel:

destination: helloexchange

#group: order

binder: rabbit1

binders:

rabbit1:

type: rabbit

environment:

spring:

rabbitmq:

host: localhost

port: 5672

username: guest

password: guest

server:

port: ${PORT:2222}

So, we have spring.cloud.stream.bindings where we binding our hellooutchannel and giving destination to helloexchange and providing type of binder i.e. rabbit1. Now inside binders we are defining rabbit1 type to be rabbit and inside environment we provided details to which required to connect RabbitMQ.

If you want to use Kafka in future you just need to change configuration in YML file and just need to do changes in pom.xml.

Now to generate class we need to enable binding. So, we annotated ProducerApplication class with @EnableBinding

@SpringBootApplication

@EnableBinding(Source.**class**)

**public** **class** ProducerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ProducerApplication.**class**, args);

}

}

Now we have a controller where we have defined a rest service where using @Autowired we have injected our Source.class

@RestController

**public** **class** GreetingsController {

**private** **static** Logger *LOG* = LogManager.*getLogger*(GreetingsController.**class**);

@Autowired

**private** Source source;

@GetMapping("/greet")

**public** Greeting createGreeting(String message) {

Greeting greeting=**new** Greeting(message);

*LOG*.info("Sending greeting!! ");

source.myoutput().send(MessageBuilder.*withPayload*(greeting).build());

**return** greeting;

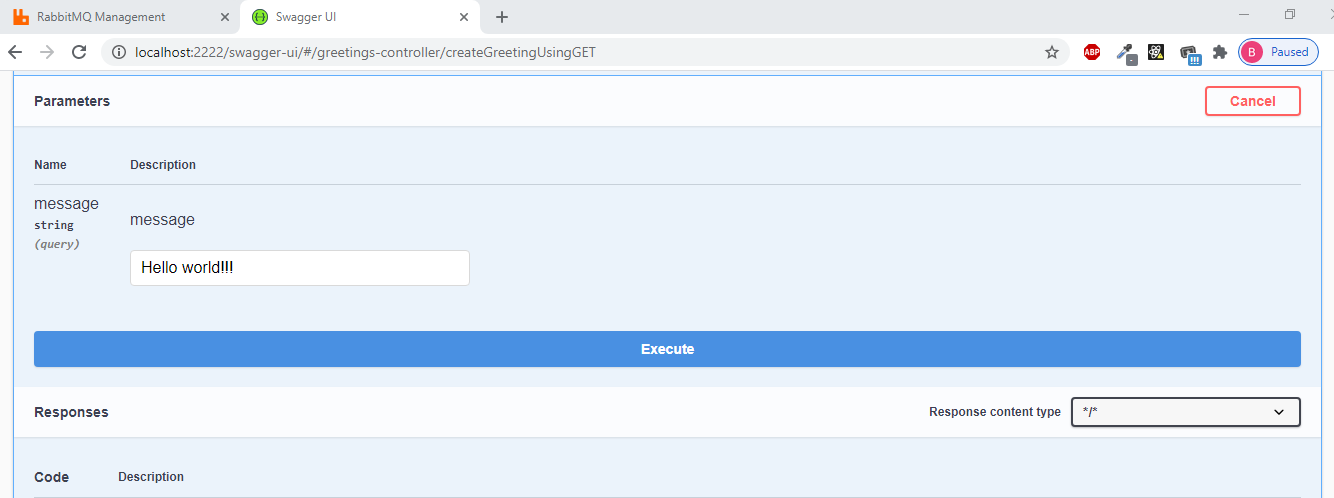
}

}

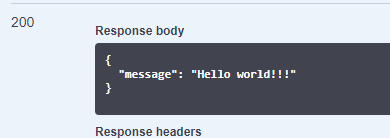
Using source object we are calling myoutput method which will return MessageChannel and using MessageBuilder we are making payload using greeting object.

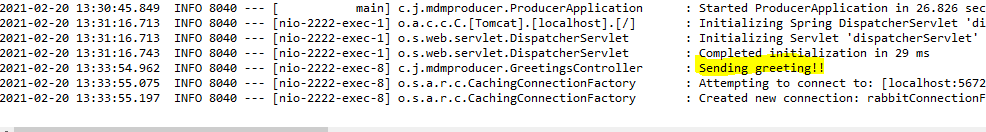
Now let start RabbitMQ using docker we already done necessary changes. And then start producer application. We have also added swagger-ui which is helpful for calling any rest service.

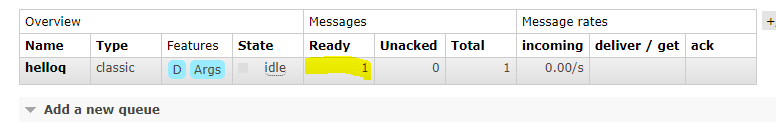
1. Call /greet service.

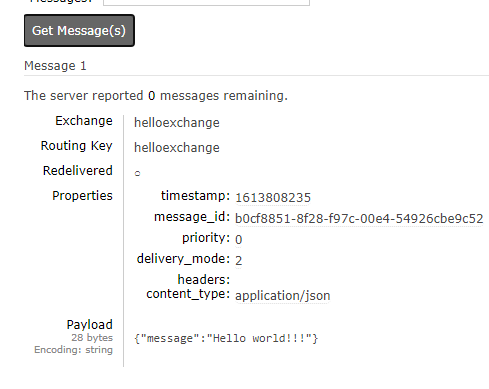


1. Click on execute you will see below message.









1. Go to RabbitMQ click on queue you will find one message in ready when you click on queue -> get message you can see the message.

Consumer-App:

1. same steps need to be followed, add RabbitMQ dependency in pom.xml

<dependency>

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

<artifactId>spring-cloud-starter-stream-rabbit</artifactId>

</dependency>

1. Now create a source class we below code. Here we are subscribing that’s why we used subcribechannel and it is incoming message so we annotated it with @Input

**public** **interface** Source {

@Input("helloinchannel")

SubscribableChannel myinput();

}

1. Now in YML file we need to put necessary config details as shown below.

spring:

cloud:

stream:

bindings:

helloinchannel:

destination: helloexchange

group: helloq

binder: rabbit1

binders:

rabbit1:

type: rabbit

environment:

spring:

rabbitmq:

host: localhost

port: 5672

username: guest

password: guest

server:

port: ${PORT:2225}

1. In Application class annotate it with @EnableBinding create a method which is annotated with @StreamListener, Whenever a message come to queue this method get called automatically.

@SpringBootApplication

@EnableBinding(Source.**class**)

**public** **class** ConsumerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ConsumerApplication.**class**, args);

}

@StreamListener("helloinchannel")

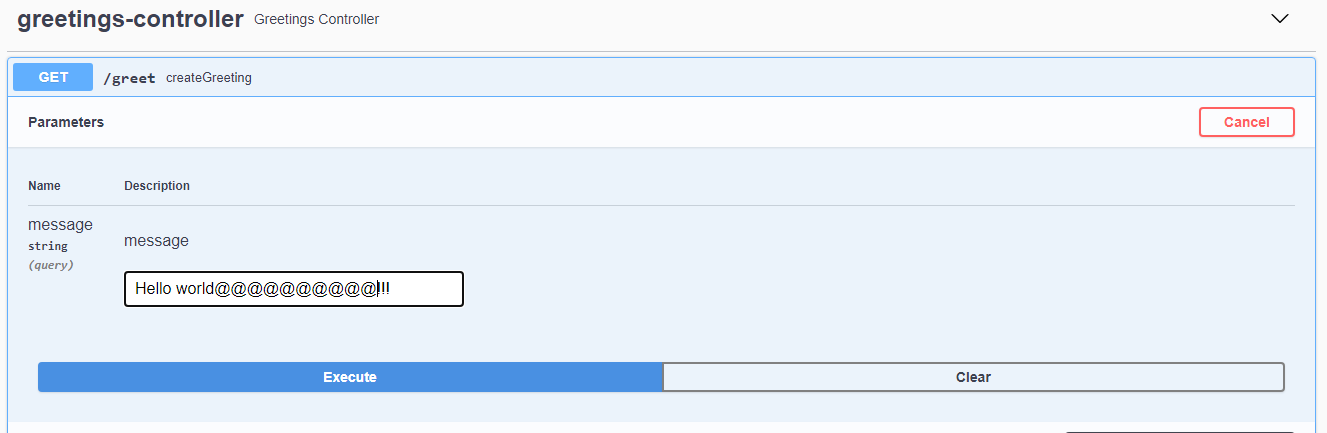
**public** **void** processGreeting(Greeting greeting) {

System.***err***.println("Recevied Greeting with message ===="+greeting.getMessage());

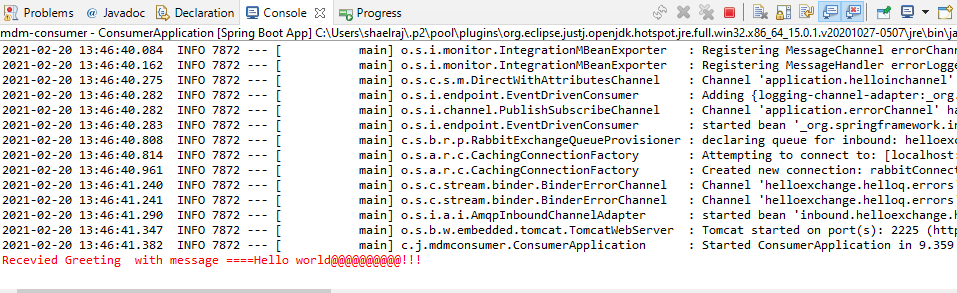
}

}

1. Start consumer application.
2. If you go to RabbitMQ you will find our helloexchange.helloq in Queue.
3. Now send a message using producer application.



1. You can see below output in console of consumer application.

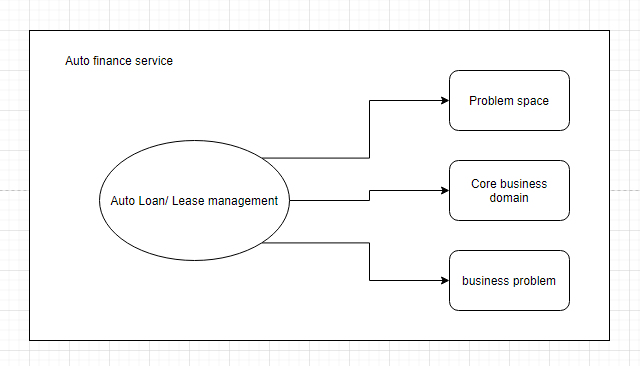


**Domain Driven Design:**

1. Problem space/ Business Domain:

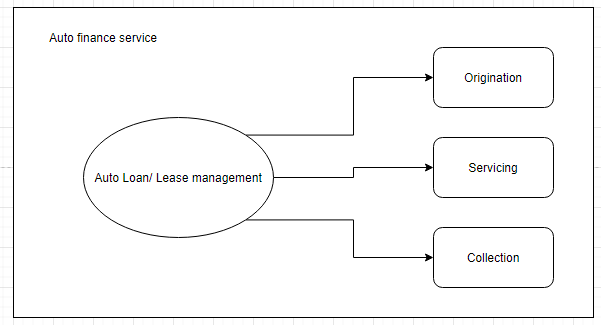
The problem space/ business domain is the starting point of the DDD journey, and it identifies the main business problem that we intend to solve using DDD.

e.g. Suppose we want to implement an application which deals with any Auto finance / loan for auto purchase. So here our problem space will be Auto Loan.



1. **Sub-Domains**:

The identification of the sub-domains essentially involves the breaking down of the various business capabilities of your main business domain into cohesive units of business functionalities.



**Origination sub-domains**: It takes care of the business capability of issuing new auto loans/lease to customers.

**Servicing Sub-Domain**: It takes care of the business capability of servicing (e.g. monthly billing/payments) these auto loans/leases.

**Collections Sub**-**Domain**: It takes care of the business capability of managing these auto loans/leases if something goes wrong e.g. customer defaults on payment.

1. **Bounded Contexts**: It is design solutions to out identified business domains/sub-domains. The identification of bounded contexts is governed primarily by the cohesiveness that you need within the business domain and between your sub-domains. In simple word it is design solution to your problem.

In Auto finance business domain, we could choose to have a single solution for the entire domain, that is a single bounded context for all the sub domains or we could choose to have a bounded context mapped to a single sub-domain/ multiple sub domain. In generally in Microservices we are defining one bounded context to each sub-domain.

1. **Domain Model**: It is the implementation of the core business logic within a specific bounded context. In business language, it involves identifying.

* Business Entities
* Business Rule
* Business flows
* Business Operations
* Business Events

In technical language within the DDD world this translates into identifying

* Aggregates/Entities/ Value Objects
* Domain Rules
* Sagas
* Commands/Queries
* Events

1. Aggregates/Entity Objects: It is central business object within your bounded context ad defines the scope of consistency within that bounded context.

Aggregate = principal identifier of your bounded context

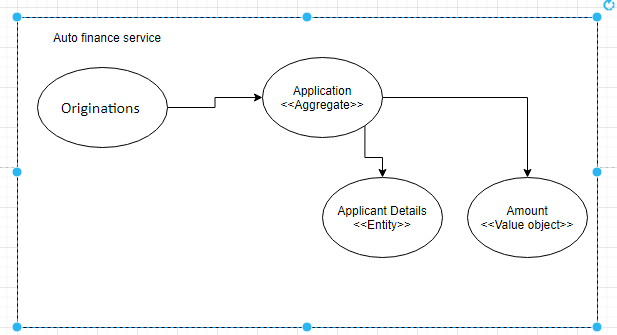
Entity Object have an identity of their own but cannot exist without the root aggregate, that is, they are created and are destroyed when the root aggregate is destroyed

Entity Objects= Secondary identifiers of your bounded context.

e.g. Order management

we can have Order class and to store order data we have table inside DB, similarly we have order Items which also have table inside DB. So now Order class contain List of order items. Now suppose we want to add one item to my order item. What will be do we can create addItem method which take order item as parameter which save data in list of order and persist all the states inside the DB. It is not a good idea to directly add order item in DB so what will we do we get order object and then in list we will add new order item and then persist whole object. Now either all data saved or all rejected. So here we can say Order is Aggregate or main class. So now it is better to call addItem with order object because we may have some business rule. We can take example of Zomato you can add item from different restaurant in same order.

Value Object have no identity and are easily replaceable within an instance of a root aggregate or an entity.



The loan Application Aggregate is the root aggregate within the originations bounded context. Without a loan application nothing exists within this bounded context.

The loan application details entity object captures the applicant details for the loan application. It has identifier of its own but cannot exist without the loan application, that is when the loan application is created the loan applicant details are created.

**Domain Rules**: It is pure business rule definitions.

Modeled as objects too, they assist the aggregate for any kind of business logic execution within the scope of a bounded context. Within our Originations bounded context a good example of a domain rule is a “state applicant compliance validation business rule. The rule basically states that depending upon the state of the loan application, additional validation checks could be applicable to loan applicant.

**Commands/Queries:**  it represents any kind of operations within the bounded context which either affect the state of the aggregate/entity or query the state of the aggregate/entity.

e.g. suppose we need to open a loan account it will change the state of Applicant details i.e. called commands. Similarly, we want to view the state using queries.

**Events**: It is means by which one bounded context notifies another bounded context. e.g. loan account opened, loan approved, cancelled etc.

**Sagas:** Sagas are business process/ workflows within your business domain. It react to multiple business events across bounded contexts and “orchestrate the business process” by coordinating interactions among these bounded contexts.

**E.g. Cargo Tracker application using DDD.**

In the cargo tracker domain, we have four main business areas:

* Booking
* Routing
* Handling
* Tracking

Booking sub-domain: this area covers all aspects of cargo booking including the following:

* Booking of cargos
* Assigning of routes to cargos
* Modification of cargos
* Cancellation of cargos

Routing sub-domain: this area covers all aspects of cargo itinerary including the following

* Optimal Itinerary allocation for cargos based on their route specification
* Voyage maintenance for the carrier’s that will carry cargos (e.g. addition of new routes)

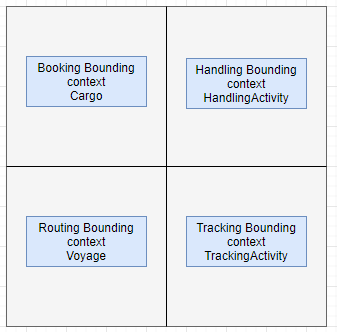
Handling: As the cargo progresses along its assigned route, it will need to be inspected/handled at the various ports of transit. This area covers all operation related to the handling activity of cargos.

Tracking: customers need comprehensive, detailed and up-to-date information of their booked cargos. The tracking business area provides this capability.

The bounded context’s domain model is the foundational piece of any DDD based architecture and is used to express the business intent of the bounded context. Identification of the domain model involves two main sets of artifacts:

* Core domain model: Aggregates, Aggregates identifiers, entities and value objects
* Domain model operations- commands, queries and events

Aggregates are responsible for capturing all state and business rule associated with the bounded context.

Each aggregate needs to be identified by uniquely called aggregates identifier. The aggregates identifier is implemented using business key like cargo🡪 bookingId, voyage🡪VoyageNumber, handlingActivity🡪 ActivityId, TrackngActivity🡪 TrackingId.

Next step to find out entities, Entities in bounded context have an identity of their own but it can’t exist without aggregates.

Within the cargo aggregates, as part of the booking process, the booking clerk needs to specify the origin of the cargo. This is mapped as an entity object that is **Location** which clearly has an identity of its own but also cannot exist on its own without the cargo aggregate.

Value Objects: They have no identity of their own and they are replaceable within the cargo aggregate and thus modeled as value objects. That is thumb rule for identifying value objects.

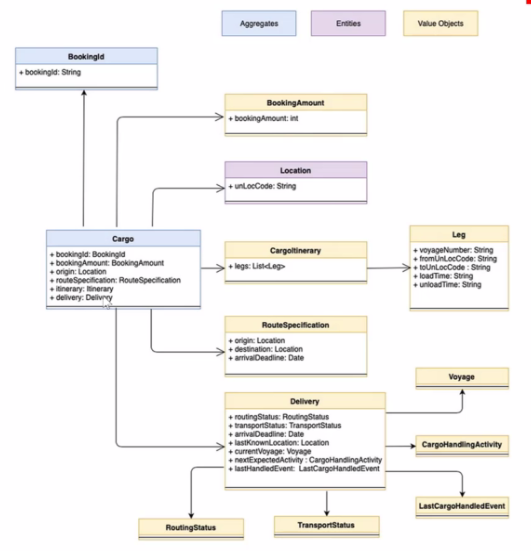
The cargo aggregate has following value objects:

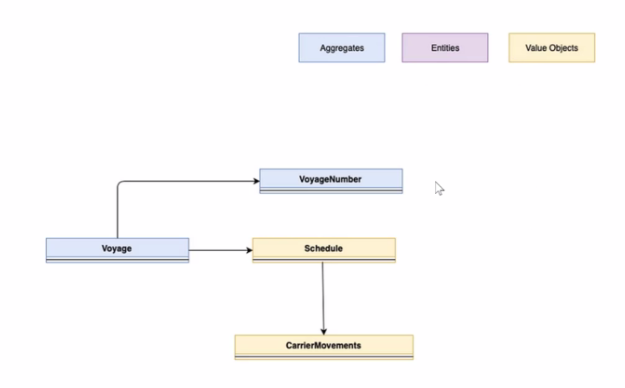
* Booking amount of the cargo
* Route specification (Origin location, destination location, Destination arrival deadline).
* Itinerary that the cargo is assigned to based on the route specification. The itinerary consists of multiple legs that the cargo might be routed through to get to the destination.
* Delivery process of the cargo against its route specification and itinerary assigned to it. The delivery progress provides details on the Routing status, Transport status, Current voyage of the cargo, last known location of the cargo, next expected activity and the last activity that occurred on the cargo.

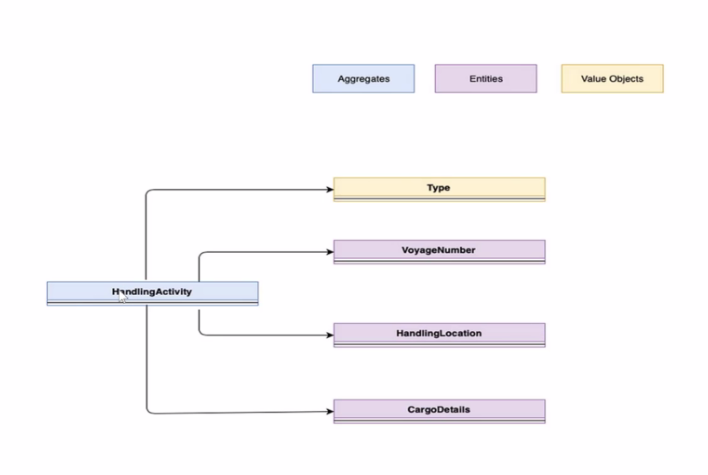
When a new cargo is booked, we will have a new route specification an empty cargo itinerary, and no delivery progress

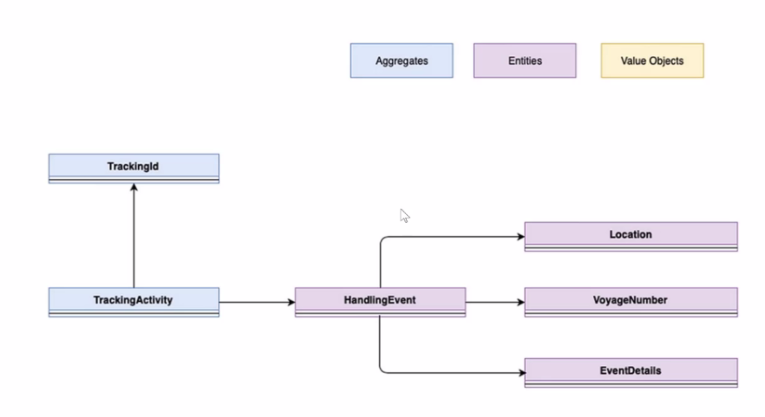
As the cargo is assigned an itinerary the empty cargo itinerary is replaced by an allocated cargo itinerary. As the cargo progress through multiple ports as part of its itinerary the delivery progress is updated ad replaced within cargo aggregate.

If customer chooses to change the delivery location of the cargo or the deadline for delivery, the route specification changes, a new cargo itinerary will be assigned, the delivery is recalculated and the bookin amount changes.



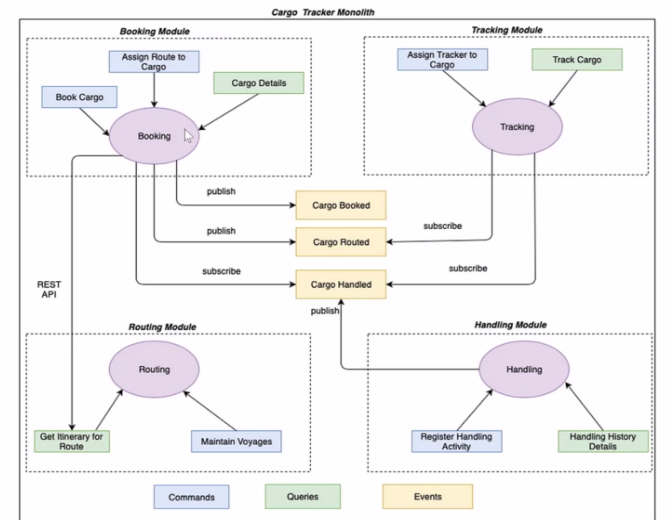






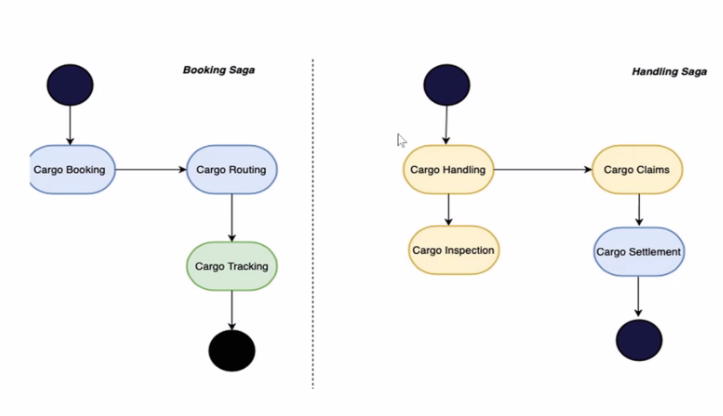
Operation within bounded context might be

* Commands that request changes of state within the bounded context
* Queries that request the state of bounded context
* Events that notify the state change of bounded context



Sagas:

The distributed nature of microservices application requires us to implement a mechanism to maintain data consistency for use cases that may span across multiple microservices. Sagas help us implement that, it can be implemented in two way either via event choreography or via event orchestration.



Domain Model Services:

It is used for two primary reasons.

* The first is to enable the bounded context’s domain model to be made available to eternal parties through will defined interfaces
* The second is interacting with external parties be it to persist the bounded context’s state change events to external message brokers, or communicate with other bounded contexts.

There are three types of domain model services for any bounded context:

* Inbound services: where we implement well defined interfaces which enable eternal parties to interact with the domain model
* Outbound services: where we implement all interactions with external repositories/ other bounded contexts.
* Application services: which act as the façade layer between the domain model and both inbound and outbound services.