Introduction to Event-driven microservices

We are into a new section and inside this section we are going to discuss about challenge number 10.

Let me try to reveal what is this challenge about?

This challenge is about building **event driven microservic**es.

So, let us try to understand more details about what are this event driven microservices and why should we build this event driven microservices.

For the same, I am going to ask you a few questions.

The very first question that I am going to ask you is, how can we avoid **temporal coupling** whenever possible inside our microservice network?

You might have heard loose coupling but temporal coupling might be a new word for you.

First, let me try to explain **what is loose coupling and how it is different from temporal coupling?**

As part of loose coupling, we will try to build our application's business logic in a separate microservices, so that they can be developed, deployed, and scaled independently.

That is what we are trying to achieve from the very start of this course, we separated all the logic

related to accounts into accounts microservice and similarly cards and loans related business logic,

we also separated into the respective microservices.

So, with that, what did we achieve?

We achieved loose coupling.

Now let us try to understand **what is temporal coupling?**

Temporal coupling occurs whenever a caller service expects an immediate response from a calling service before continuing its processing.

For example, think like you have microservice1 and microservice2. The microservice1 is dependent

on the microservice2.

So, **whenever microservice1 is trying to invoke the microservice2, the microservice1 is**

**going to continuously wait till it is going to get a response from the microservice2.**

**In this scenario, the microservice1 has a temporal coupling with the microservice2**.

So, any slow behaviour of microservice2 is going to have an impact on the microservice1.

That is why wherever possible, ***we need to avoid temporal coupling between our microservices***.

And this ***temporal coupling happens whenever we try to use synchronous communication between the services.***

As of now, we are following the synchronous communication with the help of REST APIs.

**How can we prevent this temporal coupling?**

So, let us try to understand the same.

***To avoid the temporal coupling, we need to use asynchronous communication wherever possible***

***inside our microservice network.***

Synchronous communication between the services is not always necessary.

In many real-world scenarios, asynchronous communication can fulfil the requirements very effectively.

So wherever possible, we need to establish asynchronous communication between the services.

I know you may have questions like **what is asynchronous communication?**

Do not worry about it.

We are going to talk in few minutes.

But before that, let me try to clarify **why we should not consider synchronous communication in all**

**type of scenarios inside the synchronous communication**, there are two approaches majorly used by everyone.

The first one is **imperative approach** and the second one is **reactive approach**.

*Inside the imperative approach, whenever a microservice1 is trying to invoke the microservice2, in such scenarios there will be a thread dedicatedly assigned to this communication and the thread*

*which is blocked for this operation is going to wait continuously for the response to come from the*

*microservice2.*

*So, this is a very plain synchronous communication approach.*

*Whereas in the reactive approach there will not be any threads blocked on the microservice1 to wait for the response from the microservice2.*

*Instead, what is going to happen is, there will be a thread while invoking the microservice2.*

*After the invocation the thread will go back to the thread pool and it will try to pick up the next*

*request that are coming towards the microservice.*

*When the response from the microservice2 is received by the microservice1 then only a thread from the thread pool is going to be assigned.*

*So, if you see here inside the reactive approach, we are trying to use the threads more efficiently*

*compared to the imperative approach.*

**Regardless of whether you are using imperative approach or reactive approach, the communication between the two microservices are going to be synchronous communication, which means until your microservice1 is going to receive the response from the microservice2, it is not going to process the next business logic.**

This kind of synchronous communication you may need for critical business scenarios inside your microservices, where you want to show the immediate response to the end users.

Suppose think like inside your bank website, the end user is trying to understand what is his current balance. So, he will click on a button which will reveal his current balance.

In such scenarios, the communication **must be synchronous** because the end user is waiting to see the response immediately on the screen.

But we should not use synchronous communication in all type scenarios.

Wherever possible, we need to leverage asynchronous communication.

**So how to build this asynchronous communication between the microservices?**

So, let us try to understand the same.

To build asynchronous communication between the microservices, we need to build event driven

microservices.

**So, what is an event?**

It is an incident that happens inside your microservices, which signifies a state transition or an

update inside your system.

Whenever an event takes place, we need to alert the concerned parties.

For example, take an e-commerce application inside Amazon website, whenever you place an order, the order microservice must notify the delivery microservice, which is deployed inside the Amazon network.

The communication between order microservice and the delivery microservice, it do not have to be synchronous communication.

Instead, the order microservice as soon as the end user made a payment and order is confirmed, it is going to trigger an event which will act as a notification to the delivery microservice.

In this scenario, the order microservice is just going to generate an event or trigger a notification.

Apart from that, the order microservice is **not going to wait** for that delivery process to complete

and the delivery microservice to give a successful response.

The order microservice responsibility is only to send a notification to the delivery microservice.

So, this is a classic example of event driven microservice.

So, if you see here, the communication is not going to happen synchronously, instead the communication is going to happen with the help of asynchronous communication.

Because the order microservice is not waiting for the successful response from the delivery microservice, instead it simply sends a notification or alert and its job is completed.

I hope you are clear.

Now you may have a question like how to build this event driven microservices.

Do not worry, I am going to help you around this.

To build the event driven microservices, we need to follow **event driven architecture**, producing

and consuming events with the help of **asynchronous communication event brokers** and in the same process, we can leverage to fascinating projects available inside the spring cloud ecosystem, which are **spring cloud function** and **spring cloud stream**.

I hope you are clear with the challenge and you are super excited about the event driven microservices.

Introduction to Event-driven models

We decided to build event driven microservices wherever possible inside our microservice network.

To build event driven microservices inside the industry there are **two majorly used event driven models available**.

The first model is, **publisher subscriber model**.

In short form, we can also call it as pub/sub model.

This model revolves around subscriptions.

Producers they will generate the events and that are distributed to all the interested subscribers for consumption.

Inside this model, once an event is received and consumed by the consumers, it cannot be replayed again and again, which means any new subscribers joining later will not have access to the past events.

The second model is, **event streaming model**.

Inside this model also the producers will be producing the events and the consumers will be consuming the events.

But there is a clear difference between these two models.

So, let us try to understand what is this difference.

Inside event streaming model events will be written into a log in a sequential manner.

Producers publish the events as they occur and these events are stored in a well-ordered fashion.

Now coming to the consumers instead of subscribing to the events, the consumers, they will have the ability to read from any part of the event stream, which means the events can be replayed, allowing the clients to join at any time and receive all the past events as well.

***So, this is one of the major differences and you need to choose one of them based upon your business scenario.***

***If your business scenario is not expecting your consumers to read the past events, then pub-sub***

***is an ideal model.***

***Whereas if you want to give flexibility to your consumers or subscribers to read the past events or***

***to replay the past events also, then definitely in such scenarios you need to choose event streaming***

***model.***

If needed inside event streaming model we can disable the replaying of the events feature, so there is no good or bad inside these two approaches.

Please choose based upon your business scenarios.

Now these are the two models but you may have a question like how can I follow these models or how can I implement these models inside my microservices.

For the same both models, they have various products available.

If you see here, the **pub/sub model** is frequently implemented with the RabbitMQ.

So, this is a very popular option.

On the other hand, Apache Kafka is a robust platform widely utilized for **event streaming processing**.

Inside this course also, we are going to leverage these products since these are two different products, completely different from each other, providing various capabilities.

What we are going to build using a pub-sub model

Inside this lecture, let me try to explain you, what is the scenario that we are going to build with the help of pub/sub model (RabbitMQ).

As of now, we have accounts, cards, and loans, microservices and all the scenarios that we have

as of now we are leveraging synchronous communication.

We should not try to change them because the scenarios that we discussed, they are very critical to

the business.

We should not make them asynchronous.

With the current scenarios inside our microservices, my end user can perform crud operations on the accounts, cards, and loans.

Once the execution of the logic is completed, we should give an immediate response to the end user

whether the request is processed successfully or not.

That is why we have followed the synchronous communication.

But now we decided to explore the asynchronous communication and to leverage the pub/sub model inside this section.

So, to build this asynchronous communication inside our microservices network, I am going to

take a scenario.

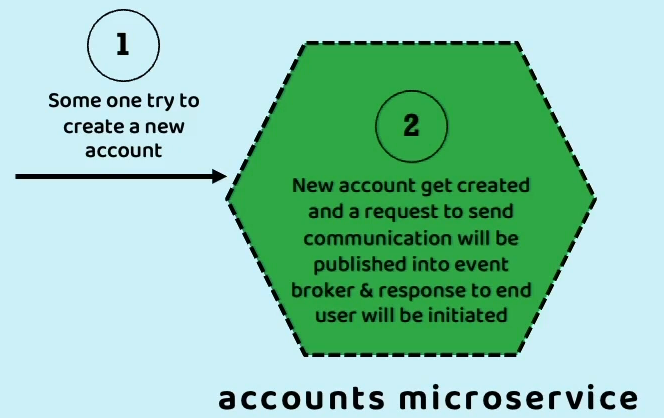
Inside this lecture, I am going to explain the same scenario. So, let us try to understand the same.

In the very first step, someone will try to create a new account with the help of accounts microservice.



So, inside our accounts microservice, we have a create API using which a new account can be created.

So, this request will be received by my accounts microservice and behind the scenes my accounts microservice is going to create the new account and at the same time it is also going to be responsible to send the communication to the end user like through SMS or through email, saying that your account is successfully created.



So, if you see here, there are two parts inside the microservice logic.

* creating a brand-new account and storing the details inside the database.
* trigger an event for a dependent microservice to send the communication.

I do not want to give my accounts microservice only to handle all the communication business logic as well.

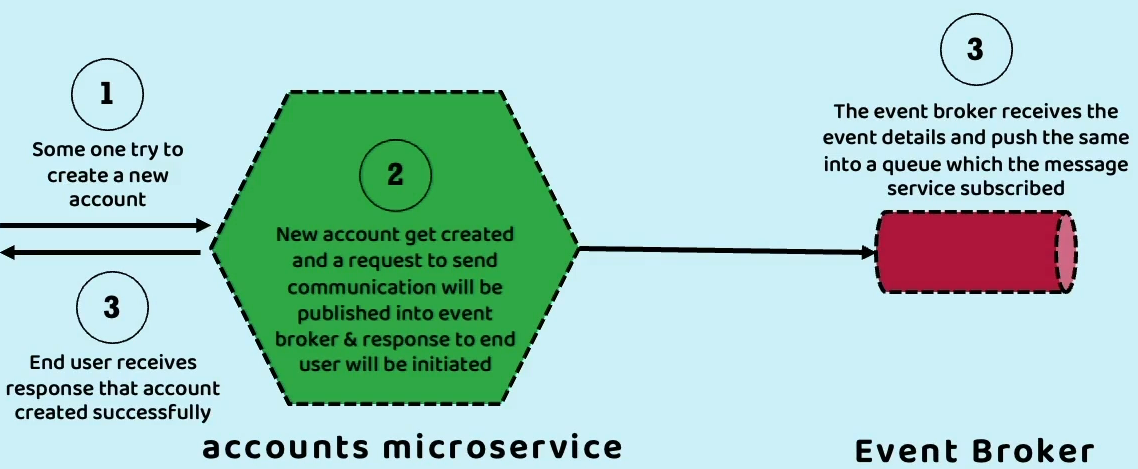
Instead, I am going to build a new separate microservice, which is going to be responsible for sending

the communication to the end user.

So that is why as soon as the account details are created successfully, my accounts microservice is

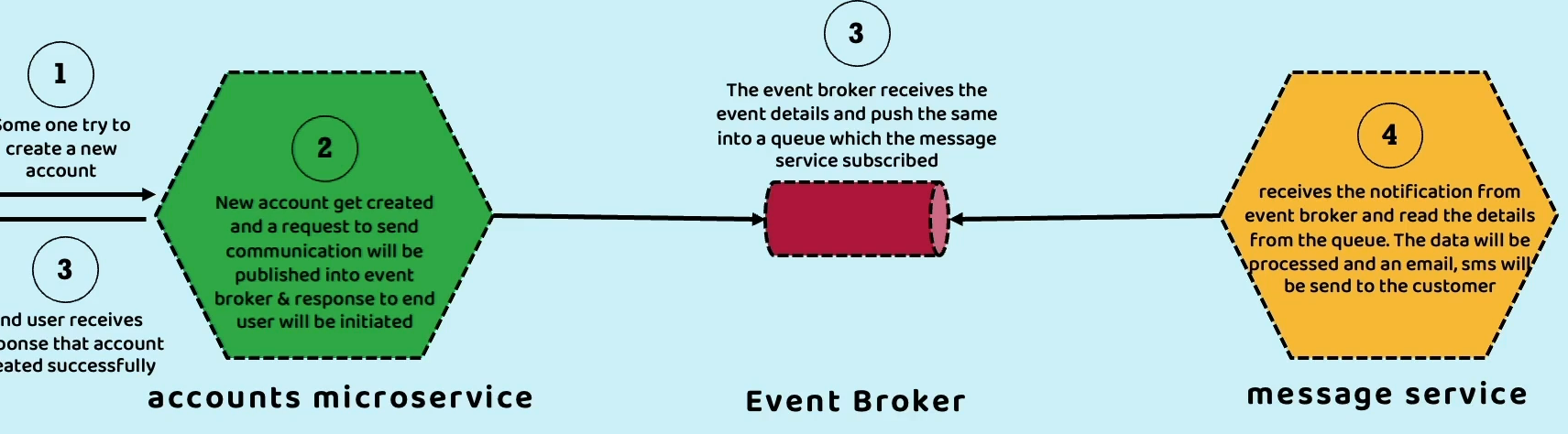
going to respond to the end user immediately saying that your account is created successfully and in

the same step3 the accounts microservice is also going to send an event to the event broker.



The event broker is going to put that event into a queue from this queue all the subscribers who subscribe to that queue will read the event details.

So, in this scenario you can see in the step 4 there will be a message microservice which will continuously monitoring the queue present inside the event broker.



As soon as the event pushed into the queue, this message microservice is going to read that event details and based upon the data present inside the event, it is going to send an email or SMS to the end user saying that your account is successfully created.

So, if you see here, there is a **middleman** sitting between two microservices.

**We call this middleman as event broker.**

**So, this event broker can be RabbitMQ in the scenario of pub/sub model or it can be Apache Kafka**

**in the scenario of event streaming model**.

I am going to discuss more about these event broker and queues in the coming lectures.

But for now, this is the sample flow that will happen during the account creation.

So, there is a synchronous communication happening in the step 1, 2, 3.

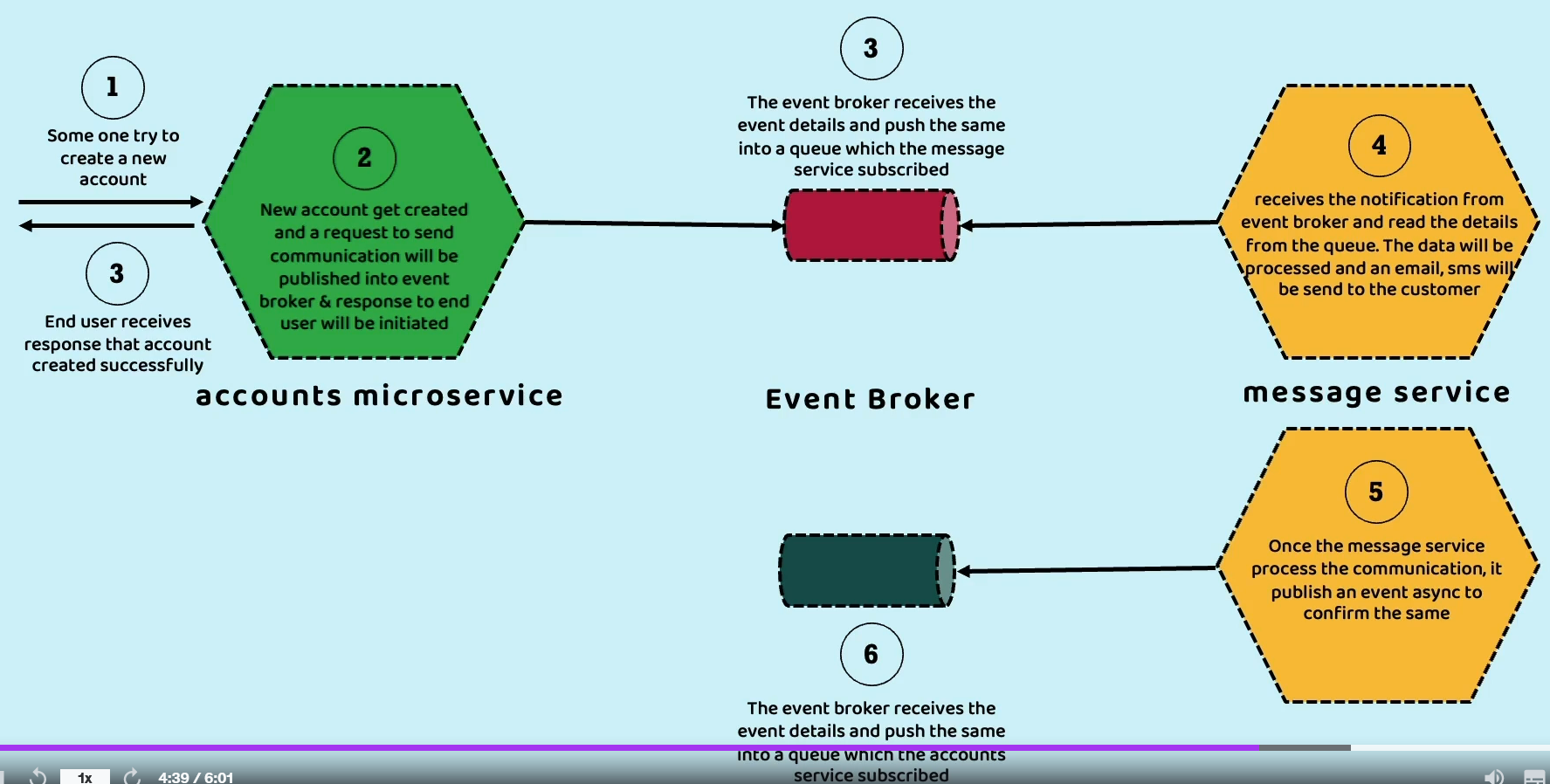
And at the same time right now we have an asynchronous scenario where we are trying to send the communication details to an end user.

So, think like after 1 minute or 2 minutes, my message microservice send the communication details to the end user in the form of SMS and email.

So now the same must be communicated back to the accounts microservice.

So that it can update that database columns saying that the communication for this account is sent to the consumer.

So that is why we are also going to have a **reverse asynchronous flow**.



So, let us try to understand the same.

So, inside the step 5, once the message service processes the communication, it is going to publish

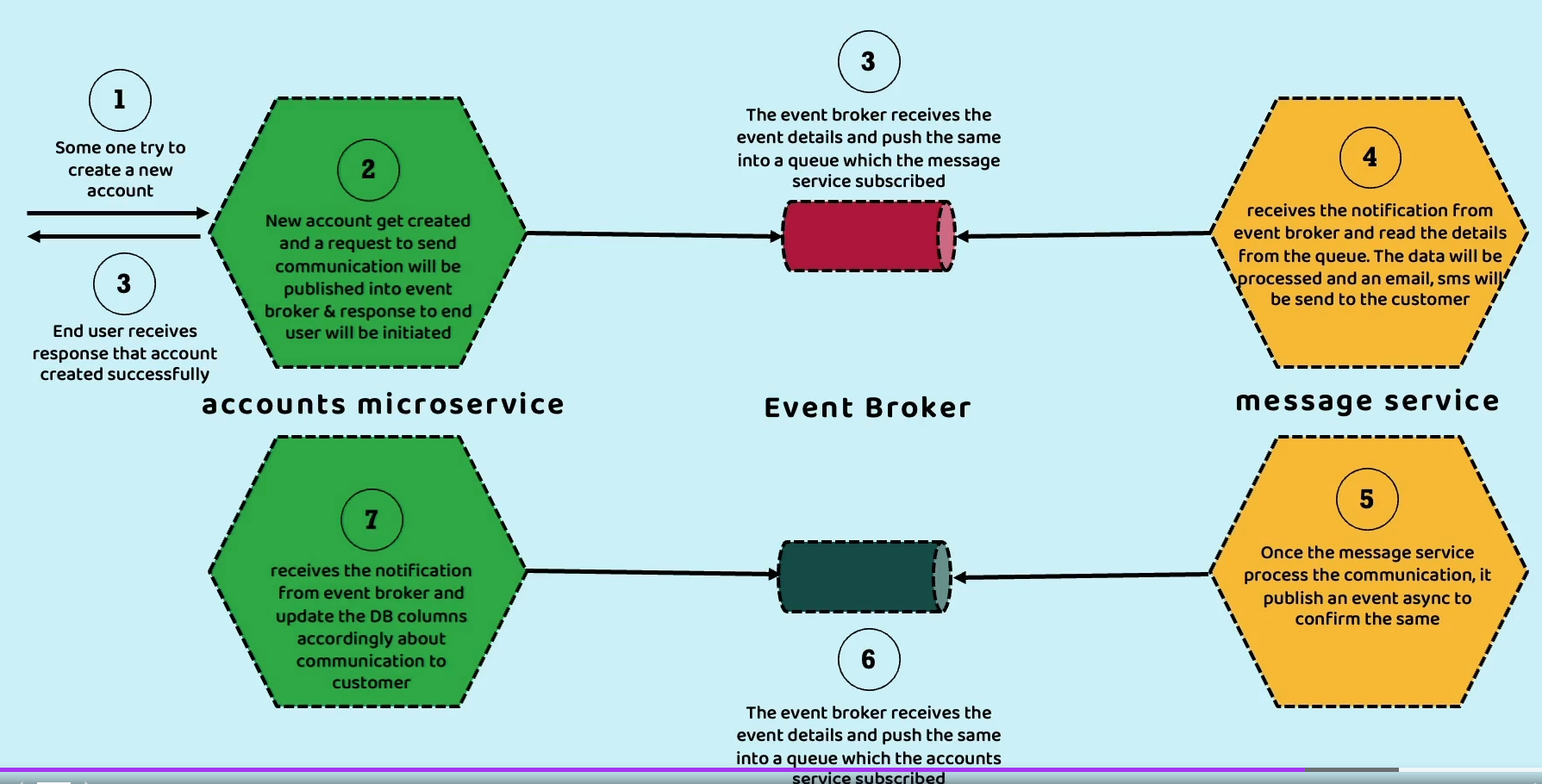
an event asynchronously confirming the same to the event broker, saying that I process the communication request and send the communication.

Now inside this event broker, there will be a queue where my accounts microservice might have subscribed.

So as soon as the message service pushes the event details to the event broker inside the step 6

accounts microservice is going to receive the notification from the event broker and inside the step7

seven my account microservice is going to update that database columns saying that communication to the customer is completed.



So here you can see the communication between message and accounts microservice is always asynchronous.

Even if my message microservice is performing slowly or even my message microservice is completely down, it is not going to impact my accounts microservice in any form.

In fact, my account microservice and message microservice, they never know from where they are receiving the data.

What is the other side of the microservice who is trying to send the data or who is trying to receive

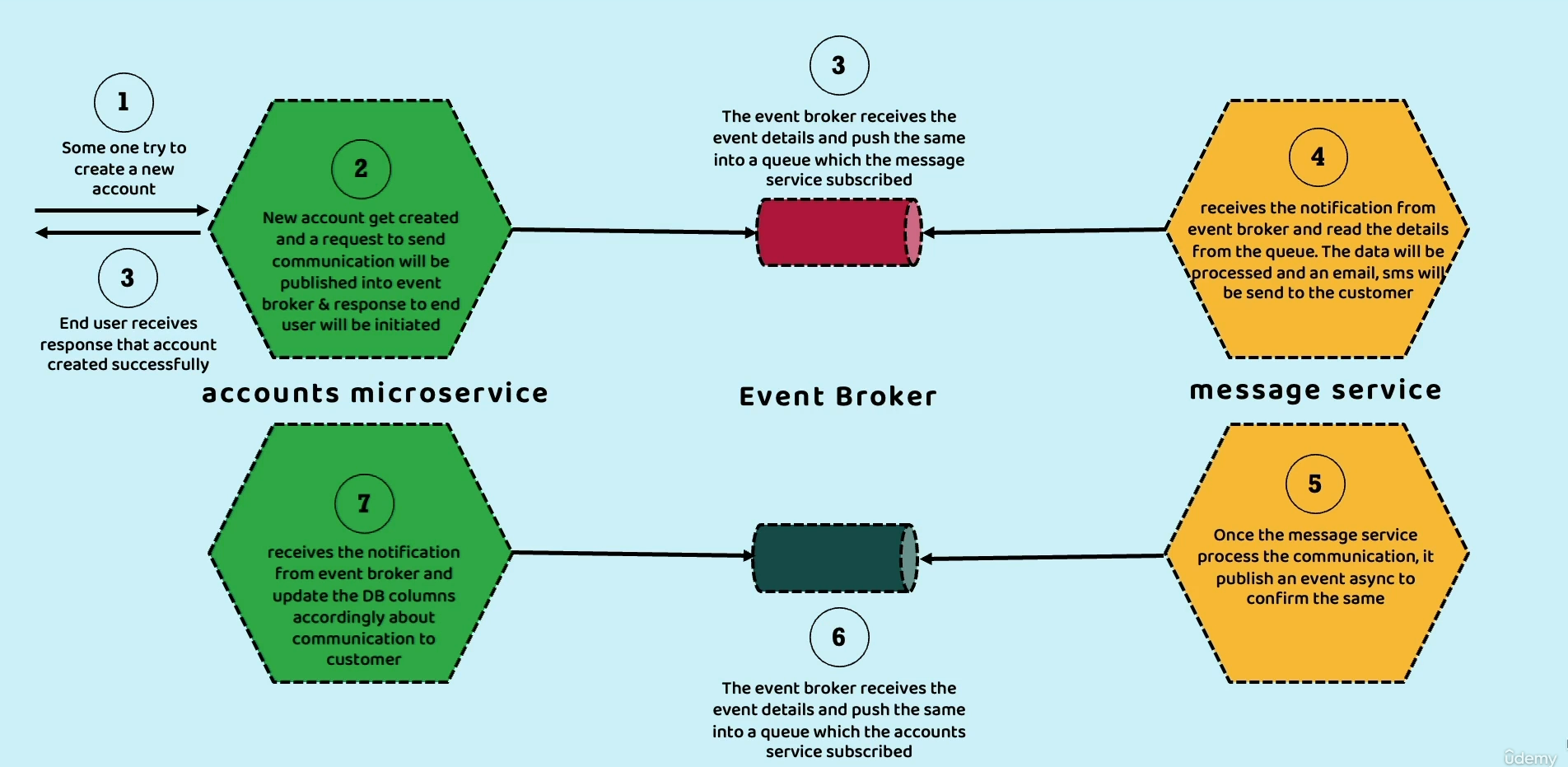
the data? They never know such details.

They simply know to push the event to the event broker or to read the event details from the event broker.

So, this is the scenario which we are going to build inside our accounts microservice and at the same time we are also going to build a new microservice with the name message.

So as an event broker inside this section, we are going to use the pub/sub model with the help of the

RabbitMQ.



Introduction to RabbitMQ

We decided to implement asynchronous communication between accounts and message microservice with the help of event broker. And this event broker can be RabbitMQ or Apache Kafka.

Inside this section we are going to leverage RabbitMQ.

Since we are going to leverage RabbitMQ inside this section, let me try to give you a very quick introduction about what is RabbitMQ, what is its internal process or jargons that we need to be aware.

RabbitMQ is an open-source message broker, which is widely recognized and utilized by majority of the companies.

So, these RabbitMQ it is going to follow the AMQP protocol.

What is AMQP, **Advanced Message Queuing Protocol** and this RabbitMQ offers flexible asynchronous messaging communication between two applications.

Previously, we discussed the major difference between pub/sub and event driven model are, inside pub/sub model, we cannot replay the events or messages whenever we use message broker like RabbitMQ.

But in the recent versions of RabbitMQ, event streaming capabilities are also provided with RabbitMQ we can also replay the events or messages.

But still majority of the people they use Apache Kafka for event streaming scenarios because by the time RabbitMQ team realizes and implement the same Apache kafka acquired most of the market.

We are also going to discuss what is the difference between RabbitMQ and Apache Kafka in the coming lectures in detail. For now, let us try to focus on RabbitMQ.

So, whenever we are using RabbitMQ, we need to be aware about few jargons or few terminologies.

The very first jargon is **producer**.

**Who is the producer?**

It is the entity or it is the service that is responsible to send the events or messages.

So, whenever we are following pub/sub model, we can also say these events as messages because RabbitMQ is a message broker.

That is why in other words, we can call events as messages whenever we are trying to use a RabbitMQ.

So, I hope you are clear.

What is producer?

Sometimes people also call this producer as **publisher** because this service is responsible to publish

the messages into the message broker.

After producer, we also have **consumer**.

**So, who is a consumer?**

It is the entity at the service that is responsible to receive the messages from the message broker

and process them.

In other words, we can also call this consumer as **subscriber** because these entities are subscribed

to the message broker to be notified whenever a message is received.

So, we have producer and we have consumer.

**So, who is going to connect them and establish a synchronous communication?**

The **message broker** is going to do that.

The **middleware** that receives messages from producers and directs them to the appropriate consumers.

We call that middleware component as message broker and RabbitMQ is one such message broker

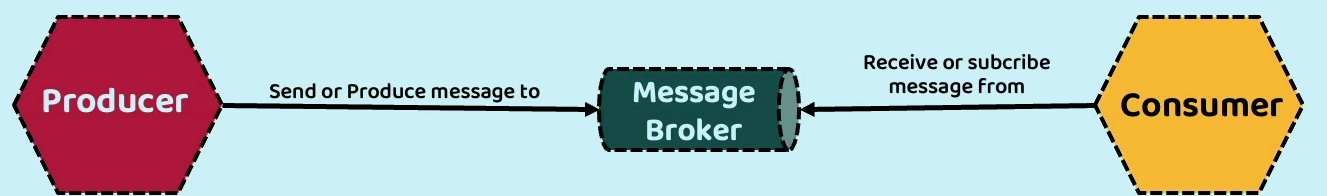
similarly, we also have many other message brokers inside the industry. But RabbitMQ is an open-source message broker, which is majorly used by most of the organizations.

That is why I am also trying to use the same.

So, if you try to visualize the relation between producer, consumer, and message broker, it is going to

look like this.

You can see on the very left-hand side we have producer and the right-hand side we have the consumer, and in between we have the message broker.



The producer will send our producers messages into message broker and the other side, the consumer who is going to subscribe to the message broker, they are going to receive the messages from the message broker whenever it is available.

**Here you may have a question like is message broker is capable of handling only one producer or one consumer?**

Off course not.

The message broker can handle any number of producers and consumers.

With that clarification, you may have one more question which is,

**how the message broker is going to identify who is the producer is trying to send the message and to which consumer it must send the message?**

For the same, let us try to understand what exactly is going to happen inside the message broker.

The messaging model of AMQP operates on two principles.

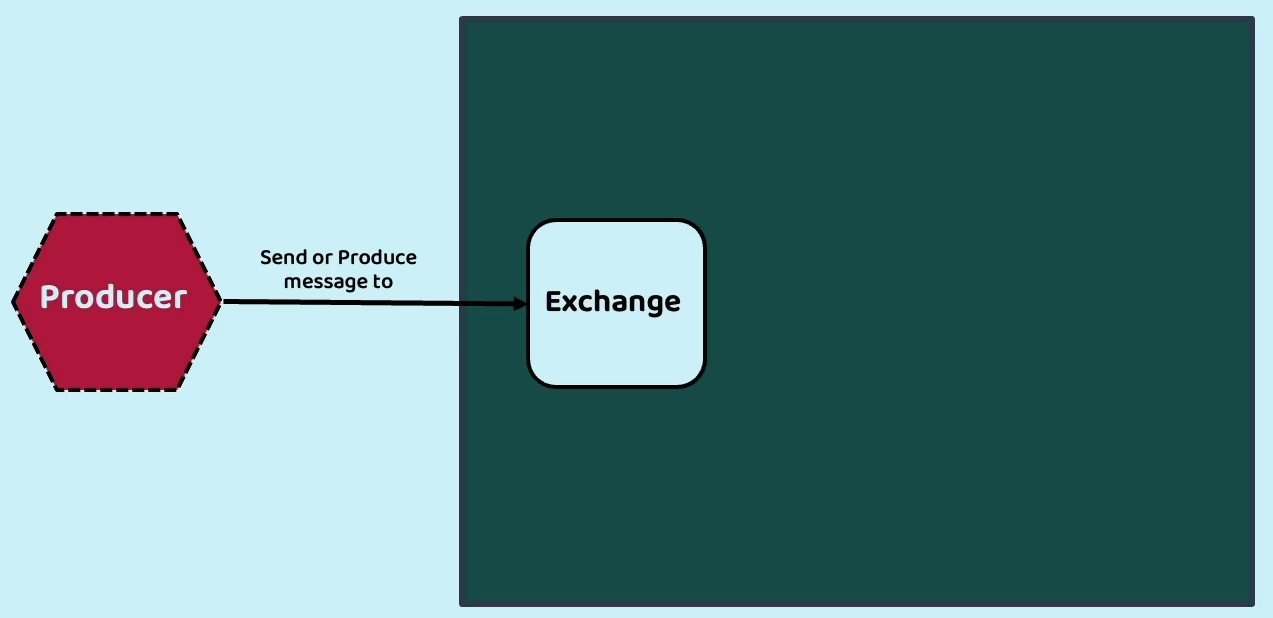
One is **exchanges** and **queues**.

So, whenever a producer wants to send a message to the message broker, it must send to the exchange present inside the message broker and based upon the specified routing rules for an exchange, the RabbitMQ is going to determine which queue should receive the copy of the message and from the queues the consumers are going to read the message.

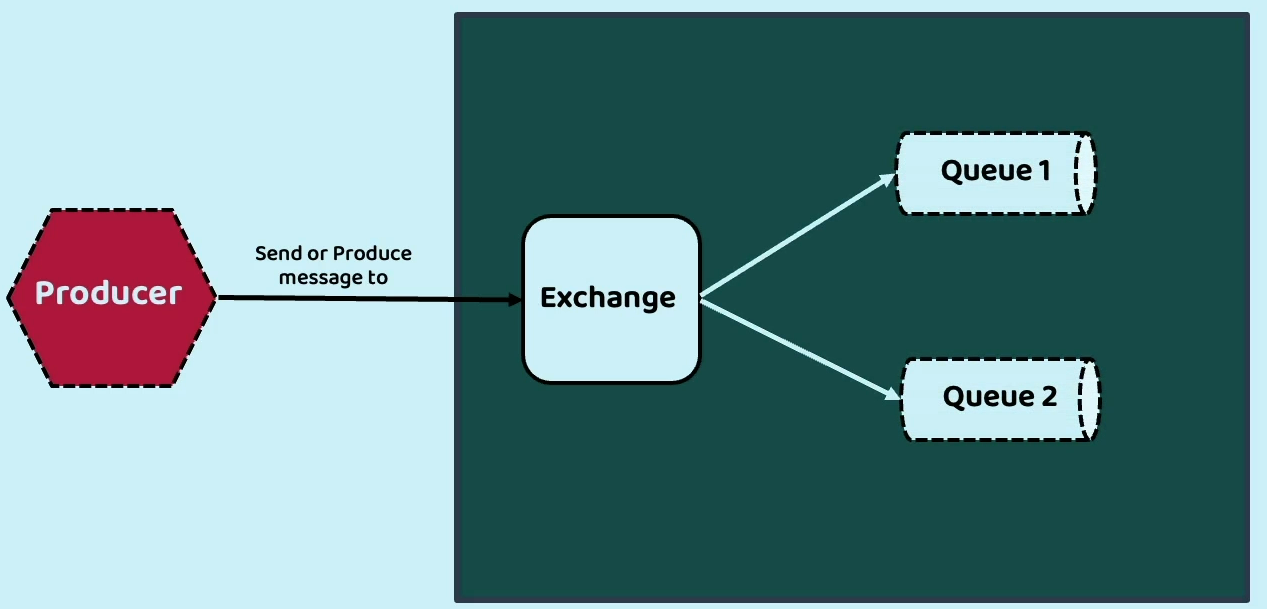
I know this may not be clear for you.

Let me try to show you the same inside a picture.

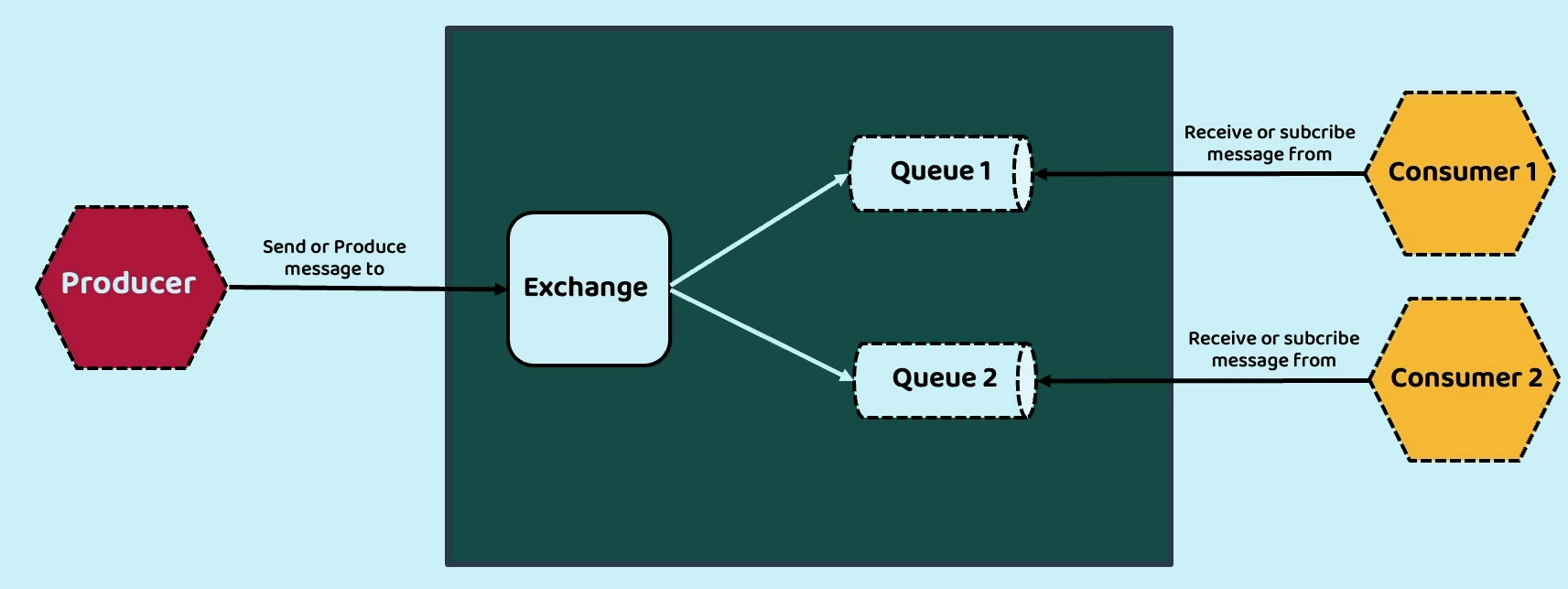
Like you can see here, first there will be a producer. Producer will always send the message to an exchange inside your message broker. This exchange is going to identify to which queue it must send the message details.



Inside your message broker you may have any number of exchanges and queues,



and to these queues the consumers are going to subscribe and whenever a message is pushed into the Queue1 or Queue2, the corresponding consumer can receive the message.



For example, the producer can send a message to the exchange and based upon the exchange details, the message can go to the Queue1 or Queue2.

If the message goes to the Queue1, then consumer one will be notified about this message as soon as it reads, the message will be deleted from the queue and there is no rule that only one consumer must be subscribed to a single queue.

Any number of subscribers can be subscribed to a queue and very similarly, a consumer can be subscribed to any number of queues.

With this, I am assuming you are clear about RabbitMQ.

Why to use Spring Cloud Function

To get started with the implementation of asynchronous communication with the help of RabbitMQ.

First, we need to create the message microservice.

This message microservice is responsible to receive the messages from the message broker and send the communication to the end users using SMS and email.

To build this message microservice we are not going to follow that traditional approach, which means I am not going to build REST services by following traditional annotations like RestController, GetMapping PostMapping.

Instead, I am going to leverage spring cloud function. So, **what is this spring cloud function** and why we need to leverage this for any event driven microservices.

So, let us try to understand the same spring cloud function, facilitate that development of business

logic by utilizing functions. So whatever business logic that my developer wants to write, he simply must write the business logic inside the functions and the rest of the infrastructural concerns is going to be taken care by the spring cloud function.

So how a developer must write the business logic by adhering to the standard function interfaces available are introduced in Java eight. These functions are namely **supplier,** **function** and **consumer**.

I know you may have too many questions right now, but let me first try to explain you what is the supplier, function, and consumer post that I am going to discuss, what are the advantages that we are going to get by building our business logic with the help of spring cloud function.

The very first function type that we have here is **supplier**.

A supplier is a function or a lambda expression that produces an output without requiring any input.

It can also be referred as a producer, publisher, or source.

That means whenever you write your business logic using supplier functional interface, then your business logic is never going to expect any input.

It is simply going to produce an event or produce an output.

In simple words for your Supplier, Function interface, there will not be any input, but there will be

an output.

So that is why we will call it as Supplier.

It is going to always supply the output without taking any input.

The very first function type that we have here is **function**.

What is a Function functional interface.

This Function accepts inputs and generate an output.

It is commonly referred as a processor because it is taking an input and it is going to process the

input and post the processing it is going to generate an output.

In simple words, your Function is going to accept an input and send an output.

The very first function type that we have here is **consumer**.

So, Consumer is a quite opposite of Supplier.

A Consumer is a function or a lambda expression that always consumes an input but does not produce any output. It can be called as a subscriber or sink.

In simple words, your Consumer is always going to have an input, but it will never be going to generate any output.

It is always going to consume, consume, but not going to give any output.

*So, these are the three different types of standard functional interfaces available inside the java8.*

*Using these three functional interfaces, only the developers are required to implement their business*

*logic with the help of functions.*

*Can you imagine any business logic that can come beyond these three functional interfaces?*

*Off course not, because we can either have a scenario where we are not taking any input but giving an output or we are going to take an input but not going to return any output.*

*So always your business logic will fall into any of these three interfaces.*

**Now coming to the question, why should you develop your business logic with the help of functions by leveraging these spring cloud function project?**

Because developing business logic with the help of spring cloud function is going to be simple and at

the same time it is going to provide you a lot of flexibility to expose your business logic using various

patterns.

By default, all your logic that you are going to write inside your functions will be exposed as a REST API automatically by your spring cloud function and if needed, you can also integrate these spring cloud function with the event brokers like RabbitMQ, Apache Kafka by adding one more project which is Spring Cloud Stream.

Now let us try to understand what more advantages we have whenever we are using spring cloud function.

- You can see here, whenever we are developing business logic with the help of functions, we can follow **reactive approach** or **imperative approach** or **hybrid approach**.

And like I said before, all your business logic is going to be implemented with the help of simple

POJO functions.

And if needed, you can compose multiple functions to give a desired output.

We are going to explore in the coming lectures how to write our business logic with the help of functions and how to club multiple functions to derive an expected output.

- And the next advantage that we have with the spring cloud functions is, like I said by default,

all your functions are going to be exposed as a http endpoint with the help of REST services

and you can also stream data from these functions by integrating them with Apache Kafka or RabbitMQ by using Spring Cloud stream framework.

So, which means the same business logic that you have written with the help of functions you can use as a REST APIs or you can also stream data by connecting them to the event brokers or if needed,

you can package these functions for standalone deployments specific to a target environment like AWS lambda, which is a serverless environment.

So, there are unlimited options that you have how you want to expose the business logic that you have inside your spring cloud function.

So, if you try to build your business logic with the REST APIs, then you are only limited to the opportunities where the REST APIs are supported.

How about streaming the data?

Or how about deploying your business logic into the serverless environment?

So, whenever your infrastructural requirements are keep changing, you need to change a lot inside your microservices if you are following the traditional approach.

Whereas with the spring cloud function, the developer will always focus on building the business logic with the help of functions and exposing that business logic with the various approach is going to be easily achieved by mentioning few properties inside application.properties file and this spring cloud function is best suitable for event driven architecture or event driven model because these functions give flexibility to us to take our business logic wherever we want.

Today we have some technology in future, if some other technology comes, we can easily migrate from one technology to other technology with a minimum configuration.

Let us try to explore the spring cloud function website.

You can see here under the spring cloud; we have a sub project with the name Spring Cloud function.

So, these are the high-level overview and what are the goals, why we need to use this spring cloud function.

So, it promotes and implements business logic via functions and decouple the development lifecycle of business logic from specific runtime target. So that the same code can run as a web point, which means REST APIs are a stream processor or a task, and it also supports running your business logic as a standalone, either in your local or in a pass environment.

Apart from that, if you can scroll down here, you will be able to see that there are many adapters

available to integrate your spring cloud functions with AWS lambda, Microsoft Azure, Apache OpenWhisk. So, all the majority of the serverless providers have adapters that we can use to integrate with the spring cloud functions.

Using this spring cloud functions, we can always build event driven microservice.

In our scenario, we are going to build a new microservice with the name message and since this microservice is going to completely use event driven model, it will be a good decision to build this message microservice with the help of spring cloud function.

From the next lecture, let us try to build our message microservice by leveraging spring cloud function.

Develop message microservice using Spring Cloud Functions – Part 1

Develop message microservice using Spring Cloud Functions – Part 2

Develop message microservice using Spring Cloud Functions – Part 3

Why to use Spring Cloud Stream

Update message and accounts microservices to stream & process the events – Part 1

Demo of Async communication or event streaming using Rabbit MQ – Part 1

Update message and accounts microservices to stream & process the events – Part 2

Demo of Async communication or event streaming using Rabbit MQ – Part 2

Demo of Async comm or event streaming using Docker container & Docker compose