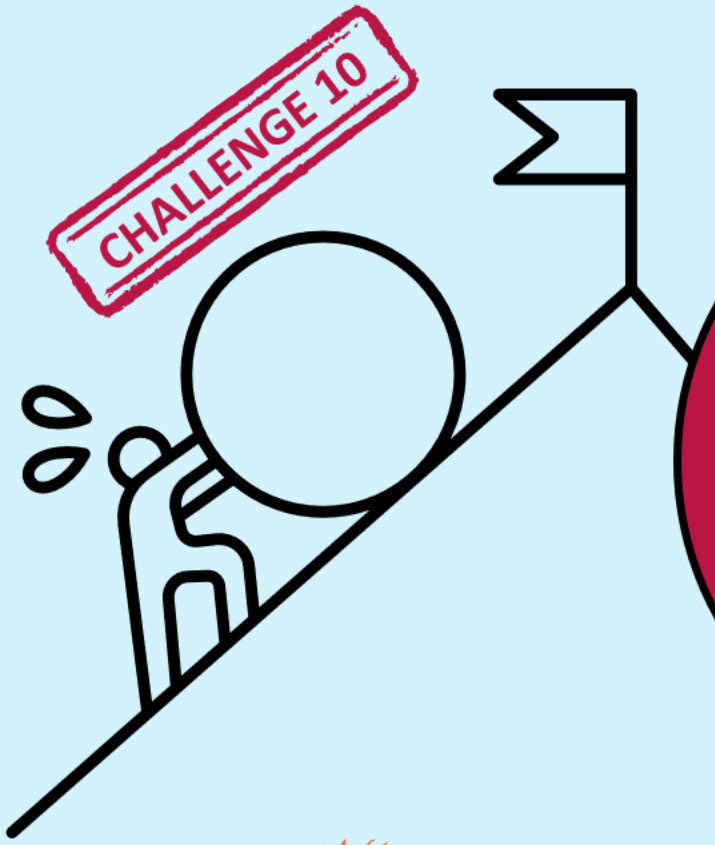
Building Event-driven microservices

eazy bytes



Avoiding temporal coupling whenever possible

Temporal coupling occurs when a caller service expects an immediate response from a callee service before continuing its processing. If the callee experiences any delay in responding, it negatively impacts the overall response time of the caller. This scenario commonly arises in synchronous communication between services. How can we prevent temporal coupling and mitigate its effects?

Using asynchronous communication

Synchronous communication between services is not always necessary. In many real-world scenarios, asynchronous communication can fulfill the requirements effectively. So, how can we establish asynchronous communication between services?

Building event driven microservices

An event, as an incident, signifies a significant occurrence within a system, such as a state transition. Multiple sources can generate events. When an event takes place, it is possible to alert the concerned parties. How can one go about constructing event-driven services with these characteristics?

Event-driven microservices can be built using Event-driven architecture, producing and consuming events using Async communication, event brokers, Spring Cloud Function, Spring Cloud Stream. Let's explore the world of event-driven microservices

Event-driven models

eazy bytes

Event-driven architectures can be built using two primary models

Publisher/Subscriber (Pub/Sub) Model

This model revolves around subscriptions. Producers generate events that are distributed to all subscribers for consumption. Once an event is received, it cannot be replayed, which means new subscribers joining later will not have access to past events.

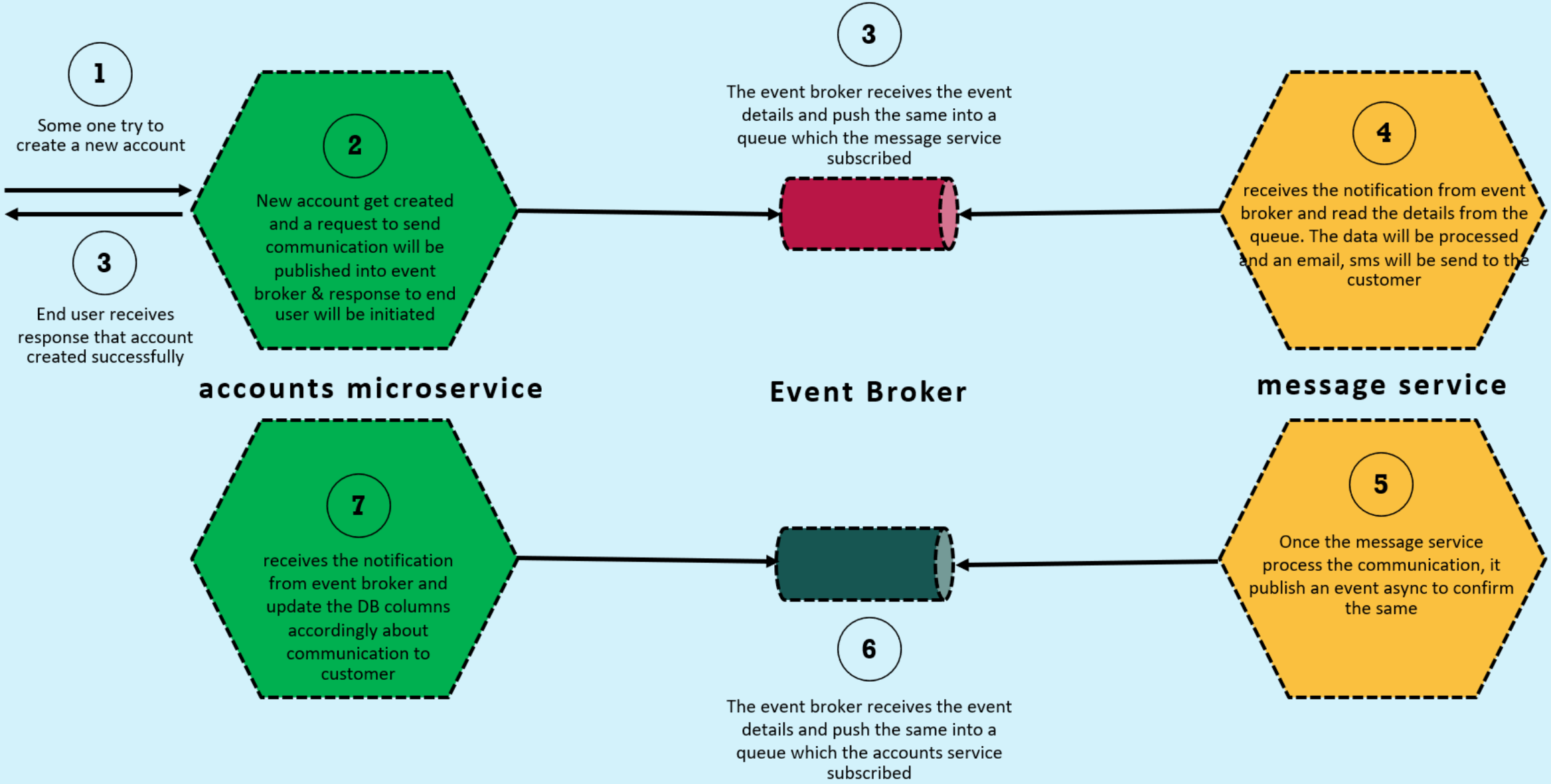
Event Streaming Model

In this model, events are written to a log in a sequential manner. Producers publish events as they occur, and these events are stored in a well-ordered fashion. Instead of subscribing to events, consumers have the ability to read from any part of the event stream. One advantage of this model is that events can be replayed, allowing clients to join at any time and receive all past events.

The pub/sub model is frequently paired with **RabbitMQ** as a popular option. On the other hand, Apache **Kafka** is a robust platform widely utilized for event stream processing.

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

eazy bytes



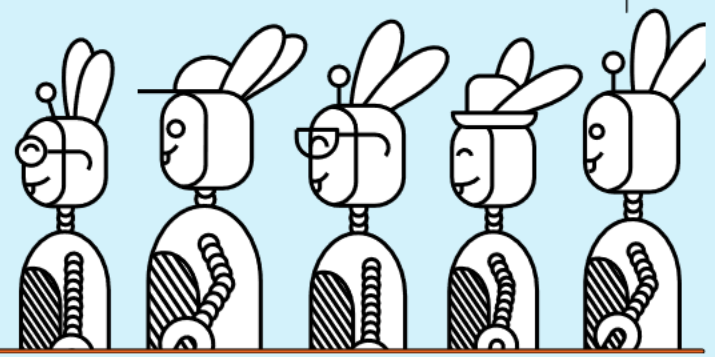
Using RabbitMQ for publish/subscribe communications

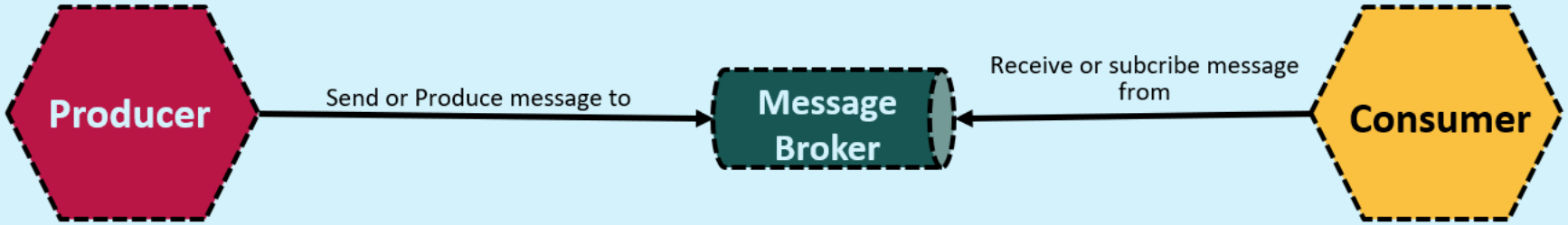
eazy bytes

RabbitMQ, an open-source message broker, is widely recognized for its utilization of AMQP (Advanced Message Queuing Protocol) and its ability to offer flexible asynchronous messaging, distributed deployment, and comprehensive monitoring. Furthermore, recent versions of RabbitMQ have incorporated event streaming functionalities into their feature set.

When using an AMQP-based solution such as RabbitMQ, the participants engaged in the interaction can be classified into the following categories:

- **Producer:** The entity responsible for sending messages (also known as the publisher).
- **Consumer:** The entity tasked with receiving messages (also known as the subscriber).
- **Message broker:** The middleware that receives messages from producers and directs them to the appropriate consumers.

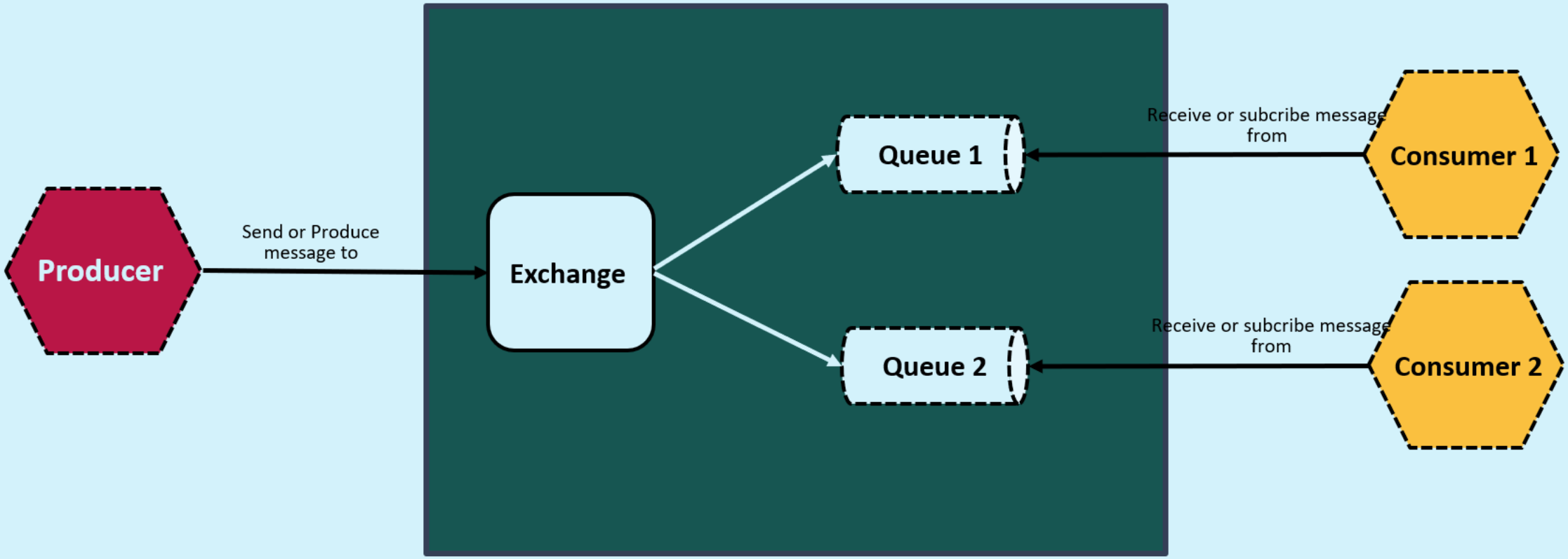




Using RabbitMQ for publish/subscribe communications

eazy bytes

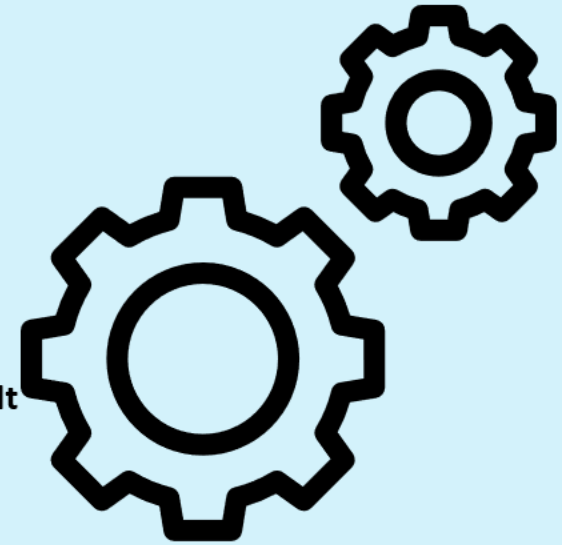
The messaging model of AMQP operates on the principles of **exchanges** and **queues**, as depicted in the following illustration. Producers transmit messages to an exchange. Based on a specified routing rule, RabbitMQ determines the queues that should receive a copy of the message. Consumers, in turn, read messages from a queue.



Why to use Spring Cloud Function ?

eazy bytes

Spring Cloud Function facilitates the development of business logic by utilizing functions that adhere to the standard interfaces introduced in Java 8, namely **Supplier**, **Function**, and **Consumer**.



- **Supplier:** A supplier is a function that produces an output without requiring any input. It can also be referred to as a producer, publisher, or source.
- **Function:** A function accepts input and generates an output. It is commonly referred to as a processor.
- **Consumer:** A consumer is a function that consumes input but does not produce any output. It can also be called a subscriber or sink.

Spring Cloud Function features:

- Choice of programming styles - reactive, imperative or hybrid.
- POJO functions (i.e., if something fits the @FunctionalInterface semantics we'll treat it as function)
- Function composition which includes composing imperative functions with reactive.
- REST support to expose functions as HTTP endpoints etc.
- Streaming data (via Apache Kafka, Solace, RabbitMQ and more) to/from functions via Spring Cloud Stream framework.
- Packaging functions for deployments, specific to the target platform (e.g., AWS Lambda and possibly other "serverless" service providers)

Steps to create functions using Spring Cloud Functions

eazy bytes

Below are the steps to create functions using Spring Cloud Functions,

1 **Initialize a spring cloud function project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the **spring-cloud-function-context** maven dependency

2 **Implement the business logic using functions**

Develop two functions with the name email() and sms() like in the image. To make it simple, for now they just have logic of logging the details. But in real projects you can write logic to send emails and messages.

To enable Spring Cloud Function to recognize our functions, we need to register them as beans. Proceed with annotating the MessageFunctions class as @Configuration and the methods email() & sms() as @Bean to accomplish this.

```java
@Configuration
public class MessageFunctions {

    private static final Logger log = LoggerFactory.getLogger(MessageFunctions.class);

    @Bean
    public Function<AccountsMsgDto, AccountsMsgDto> email() {
        return accountsMsgDto -> {
            log.info("Sending email with the details : " +  accountsMsgDto.toString());
            return accountsMsgDto;
        };
    }

    @Bean
    public Function<AccountsMsgDto, Long> sms() {
        return accountsMsgDto -> {
            log.info("Sending sms with the details : " + accountsMsgDto.toString());
            return accountsMsgDto.accountNumber();
        };
    }
```

Steps to create functions using Spring Cloud Functions

eazy bytes

3 **Composing functions:** If our scenario needs multiple functions to be executed, then we need to compose them otherwise we can use them as individual functions as well. Composing functions can be achieved by defining a property in application.yml like shown below,

```
spring:
  cloud:
    function:
      definition: email|sms
```

The property spring.cloud.function.definition enables you to specify which functions should be managed and integrated by Spring Cloud Function, thereby establishing a specific data flow. In the previous step, we implemented the email() and sms() functions. We can now instruct Spring Cloud Function to utilize these functions as building blocks and generate a new function derived from their composition.

In serverless applications designed for deployment on FaaS platforms like AWS Lambda, Azure Functions, Google Cloud Functions, or Knative, it is common to have one function defined per application. The definition of cloud functions can align directly with functions declared in your application on a one-to-one basis. Alternatively, you can employ the pipe (|) operator to compose functions together in a data flow. In cases where you need to define multiple functions, the semicolon (;) character can be used as a separator instead of the pipe (|).

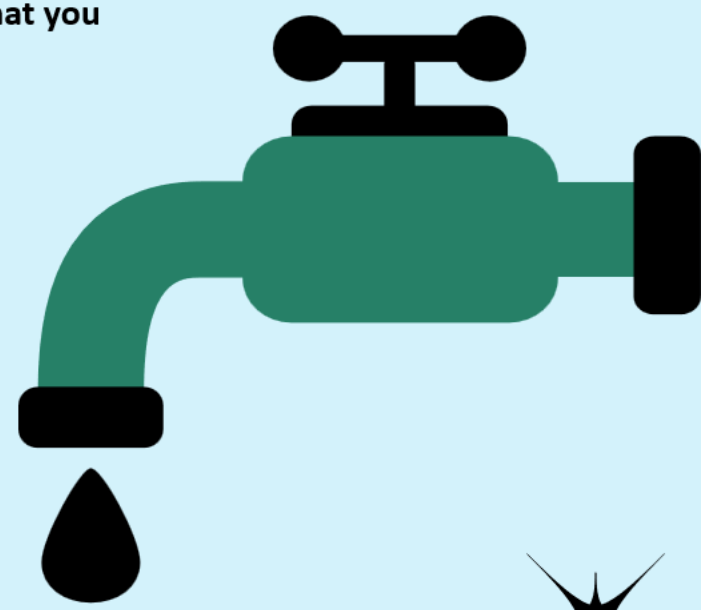
Based on the provided functions, the framework offers various ways to expose them according to our needs. For instance, Spring Cloud Function can automatically expose the functions specified in spring.cloud.function.definition as REST endpoints. This allows you to package the application, deploy it on a FaaS platform such as Knative, and instantly have a serverless Spring Boot application. But that is not what we want. Moving forward, the next step involves integrating it with **Spring Cloud Stream** and binding the function to message channels within an event broker like RabbitMQ.

Why to use Spring Cloud Stream ?

eazy bytes

Spring Cloud Stream is a framework designed for creating scalable, event-driven, and streaming applications. Its core principle is to allow developers to focus on the business logic while the framework takes care of infrastructure-related tasks, such as integrating with a message broker.

Spring Cloud Stream leverages the native capabilities of each message broker, while also providing an abstraction layer to ensure a consistent experience regardless of the underlying middleware. By just adding a dependency to your project, you can have functions automatically connected to an external message broker. The beauty of this approach is that you don't need to modify any application code; you simply adjust the configuration in the application.yml file.



The framework supports integrations with RabbitMQ, Apache Kafka, Kafka Streams, and Amazon Kinesis. There are also integrations maintained by partners for Google PubSub, Solace PubSub+, Azure Event Hubs, and Apache RocketMQ.

The core building blocks of Spring Cloud Stream are:

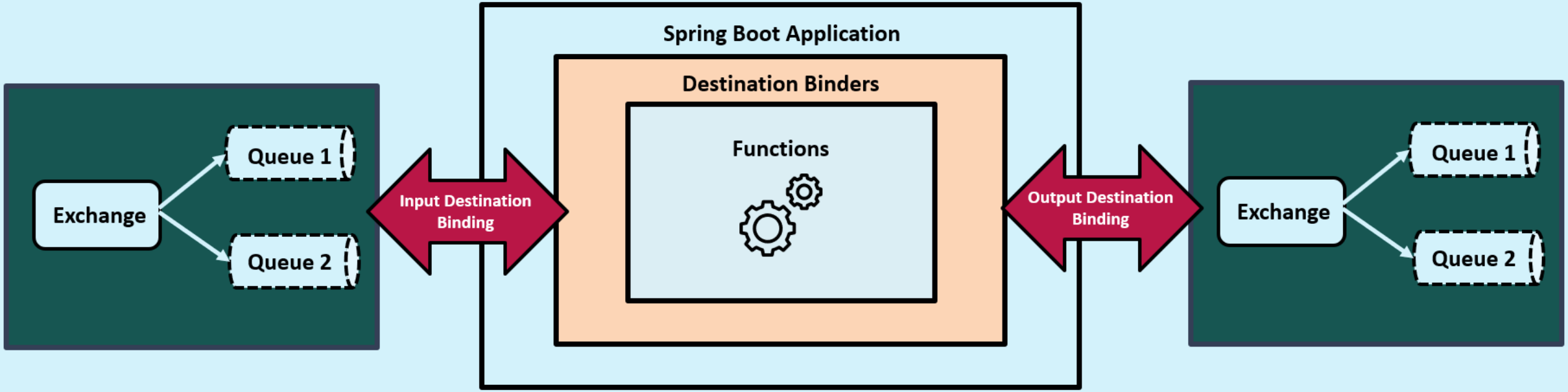
- **Destination Binders:** Components responsible to provide integration with the external messaging systems.
- **Destination Bindings:** Bridge between the external messaging systems and application code (producer/consumer) provided by the end user.
- **Message:** The canonical data structure used by producers and consumers to communicate with Destination Binders (and thus other applications via external messaging systems).

Why to use Spring Cloud Stream ?

eazy bytes

Spring Cloud Stream equips a Spring Boot application with a destination binder that seamlessly integrates with an external messaging system. This binder takes on the responsibility of establishing communication channels between the application's producers and consumers and the entities within the messaging system (such as exchanges and queues in the case of RabbitMQ). These communication channels, known as destination bindings, serve as connections between applications and brokers.

A destination binding can function as either an input channel or an output channel. By default, Spring Cloud Stream maps each binding, both input and output, to an exchange within RabbitMQ (specifically, a topic exchange). Additionally, for each input binding, it binds a queue to the associated exchange. This queue serves as the source from which consumers receive and process events. This configuration provides the necessary infrastructure for implementing event-driven architectures based on the pub/sub model.



Steps to create bindings using Spring Cloud Stream

eazy bytes

Below are the steps to create bindings using Spring Cloud Stream,

1 **Add the Stream related dependencies:** Add the maven dependencies spring-cloud-stream, spring-cloud-stream-binder-rabbit inside pom.xml of message service where we defined functions

2 **Add the stream binding and rabbitmq properties inside application.yml of message service**

We need to define input binding for each function accepting input data, and an output binding for each function returning output data. Each binding can have a logical name following the below convention. Unless you use partitions (for example, with Kafka), the <index> part of the name will always be 0. The <functionName> is computed from the value of the spring.cloud.function.definition property.

Input binding: <functionName> + -in- + <index>
Output binding: <functionName> + -out- + <index>

The binding names exist only in Spring Cloud Stream and RabbitMQ doesn't know about them. So to map between the Spring Cloud Stream binding and RabbitMQ, we need to define destination which will be the exchange inside the RabbitMQ. group is typically application name, so that all the instances of the application can point to same exchange and queue.

The queues will be created inside RabbitMQ based on the queue-naming strategy (<destination>.<group>) includes a parameter called consumer group.

```
spring:
  application:
    name: message
  cloud:
    function:
      definition: email|sms
    stream:
      bindings:
        emailsms-in-0:
          destination: send-communication
          group: ${spring.application.name}
        emailsms-out-0:
          destination: communication-sent
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
    connection-timeout: 10s
```

Event producing and consuming in accounts microservice

eazy bytes

Below are the steps for event producing and consuming in accounts microservice

1 **Autowire StreamBridge class:** StreamBridge is a class inside Spring Cloud Stream which allows user to send data to an output binding. So to produce the event, autowire the StreamBridge class into the class from where you want to produce a event

2 **Use send() of StreamBridge to produce a event like shown below,**

```
@Override
public void createAccount(CustomerDto customerDto) {
    Customer customer = CustomerMapper.mapToCustomer(customerDto, new Customer());
    Optional<Customer> optionalCustomer = customerRepository.findByMobileNumber(
            customerDto.getMobileNumber());
    if (optionalCustomer.isPresent()) {
        throw new CustomerAlreadyExistsException("Customer already registered with given mobileNumber "
                + customerDto.getMobileNumber());
    }
    Customer savedCustomer = customerRepository.save(customer);
    Accounts savedAccount = accountsRepository.save(createNewAccount(savedCustomer));
    sendCommunication(savedAccount, savedCustomer);
}

private void sendCommunication(Accounts account, Customer customer) {
    var accountsMsgDto = new AccountsMsgDto(account.getAccountNumber(), customer.getName(),
            customer.getEmail(), customer.getMobileNumber());
    log.info("Sending Communication request for the details: {}", accountsMsgDto);
    var result = streamBridge.send("sendCommunication-out-0", accountsMsgDto);
    log.info("Is the Communication request successfully processed ? : {}", result);
}
```

Event producing and consuming in accounts microservice

eazy bytes

3

Create a function to accept the event: Inside accounts microservice, we need to create a function that accepts the event and update the communication status inside the DB. Below is a sample code snippet of the same

```
@Configuration
public class AccountsFunctions {

   private static final Logger log = LoggerFactory.getLogger(AccountsFunctions.class);

   @Bean
   public Consumer<Long> updateCommunication(IAccountsService accountsService) {
      return accountNumber -> {
         log.info("Updating Communication status for the account number : " + accountNumber.toString());
         accountsService.updateCommunicationStatus(accountNumber);
      };
   }
```

Event producing and consuming in accounts microservice

eazy bytes

4 **Add the stream binding and rabbitmq properties inside application.yml of accounts service**

when accounts microservice want to produce a event using StreamBridge, we should have a supporting stream binding and destination. The same we created with the names `sendCommunication-out-0 and send-communication`

Similarly we need to define input binding for the function `updateCommunication` to accept the event using the destination `communication-sent.` So when the message service push a event into the exchange of `communication-sent,` the same will be processed by the function updateCommunication

```yaml
spring:
  application:
    name: "accounts"
  cloud:
    function:
      definition: updateCommunication
    stream:
      bindings:
        updateCommunication-in-0:
          destination: communication-sent
          group:  ${spring.application.name}
        sendCommunication-out-0:
          destination: send-communication
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
    connection-timeout: 10s
```

Apache Kafka Vs RabbitMQ

eazy bytes

Kafka and RabbitMQ are both popular messaging systems, but they have some fundamental differences in terms of design philosophy, architecture, and use cases. Here are the key distinctions between Kafka and RabbitMQ:

- **Design:** Kafka is a distributed event streaming platform, while RabbitMQ is a message broker. This means that Kafka is designed to handle large volumes of data, while RabbitMQ is designed to handle smaller volumes of data with more complex routing requirements.

- **Data retention:** Kafka stores data on disk, while RabbitMQ stores data in memory. This means that Kafka can retain data for longer periods of time, while RabbitMQ is more suitable for applications that require low latency.

- **Performance:** Kafka is generally faster than RabbitMQ, especially for large volumes of data. However, RabbitMQ can be more performant for applications with complex routing requirements.

- **Scalability:** Kafka is highly scalable, while RabbitMQ is more limited in its scalability. This is because Kafka can be scaled horizontally to any extent by adding more brokers to the cluster

Ultimately, the best choice for you will depend on your specific needs and requirements. If you need a high-performance messaging system that can handle large volumes of data, Kafka is a good choice. If you need a messaging system with complex routing requirements, RabbitMQ is a good choice.