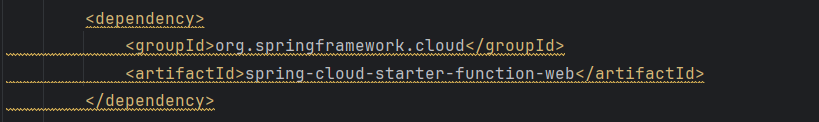
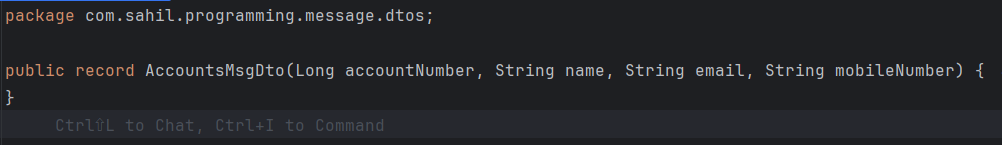
**Spring Cloud Function**

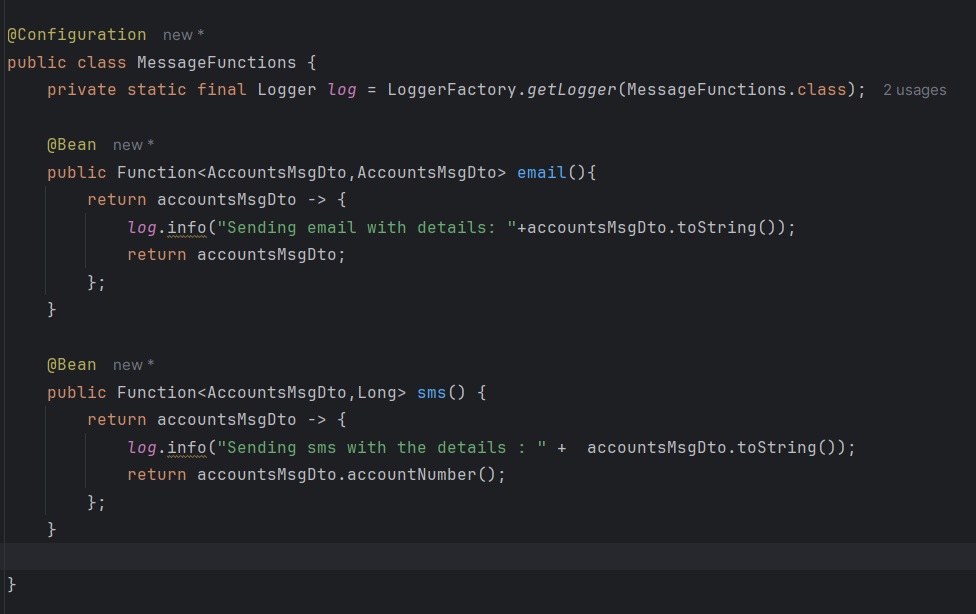
In a traditional approach, we typically create REST APIs and invoke them using HTTP methods like GET or POST. However, Spring provides an alternative functionality through Spring Cloud Function, where operations can be executed by referencing their respective function names directly. This approach expects the business logic to be implemented in the form of lambda expressions.

**How to Create a Spring Cloud Function**

**Step 1**: Add the following dependency to the pom.xml file:

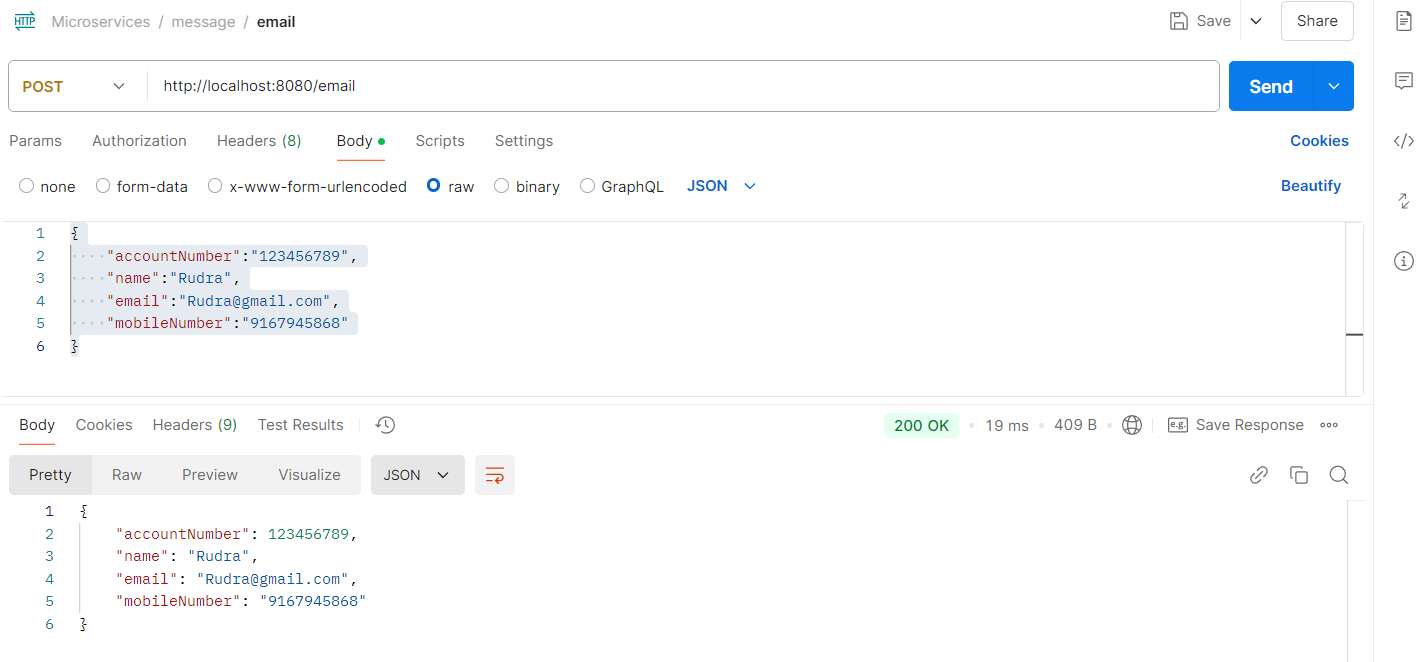


**Step 2**: Create a DTO class to accept input from the message broker and return output back to it.  
Spring Cloud Function allows you to use record for creating a DTO. A record automatically sets all variables as final and provides getters for them without the need for explicit implementation.**Step 3**: Define the Functions

Below is an example of creating lambda functions:

In the example above, we’ve defined two lambda functions, email() and sms().

* The email function uses the functional interface Function, which takes AccountsMsgDto as both input and output.
* The sms function uses the Function interface to take AccountsMsgDto as input and return a Long as output.

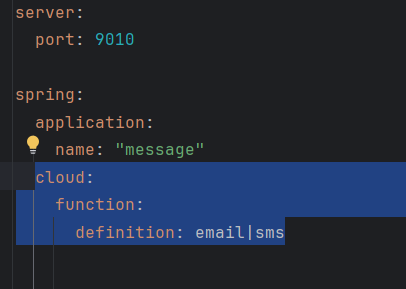


As shown in the image above, we can invoke the API directly using the function name instead of a traditional REST API endpoint.

**Composite Functions**

Now, let's consider a scenario where we want to call both the email and sms operations at the same time. To achieve this, we need to create a **composite function**. A composite function allows us to chain multiple functions together.

To set this up, we need to make changes to the application.yml file as follows:



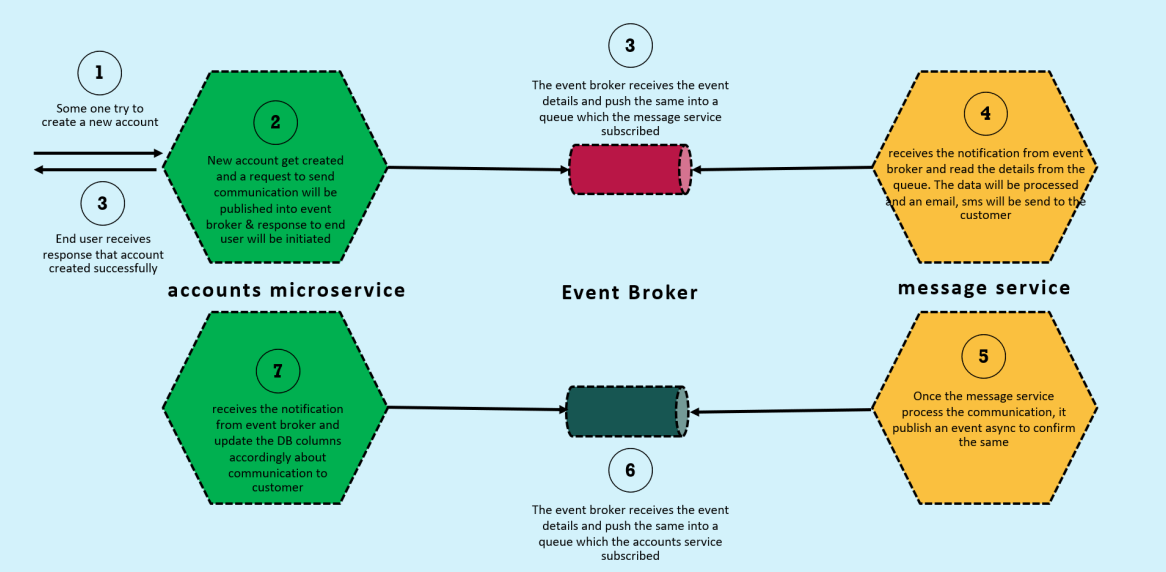
In this configuration:

* email|sms indicates a composite function where the output of the email function will be passed as the input to the sms function.
* The pipe symbol | is used to chain functions together in the desired order of execution.

With this setup, when the function is invoked, both the email and sms operations will be executed sequentially.

**Integrating RabbitMQ with Accounts and Message Microservices**

The communication flow between the **Accounts Microservice**, **Event Broker (RabbitMQ)**, and **Message Service** ensures seamless and reliable notification delivery to customers. Below is the step-by-step breakdown of the flow:

**Flow Design**

**Step 1: User Account Creation**

* A user initiates a request to create a **new account** in the **Accounts Microservice**.
* Once the account is successfully created, the system prepares to notify the customer.

**Step 2: Publish Event to Event Broker**

* After creating the account, the **Accounts Microservice** sends a **request event** to the **Event Broker**.
* Simultaneously, the user receives a response confirming that the account has been created successfully.

**Step 3: Event Broker Pushes Event to Queue**

* The **Event Broker (RabbitMQ)** receives the event details and pushes the same into a **queue**.
* The **Message Service** is subscribed to this queue.

**Step 4: Message Service Processes Notifications**

* The **Message Service** receives the notification from the queue and reads the event details.
* The service processes the data and sends an **email** or **SMS** notification to the customer confirming that the account has been created.

**Step 5: Acknowledgment to Event Broker**

* Once the **Message Service** processes the communication (email/SMS), it publishes an **acknowledgment event** asynchronously to confirm that the notification has been sent successfully.

**Step 6: Event Broker Publishes to Accounts Microservice**

* The **Event Broker** receives the acknowledgment event from the **Message Service** and pushes it into another queue, which the **Accounts Microservice** subscribes to.

**Step 7: Update Database in Accounts Microservice**

* The **Accounts Microservice** receives the acknowledgment notification from the **Event Broker**.
* It updates the database to reflect the communication status (e.g., "Email Sent" or "SMS Sent") to the customer.

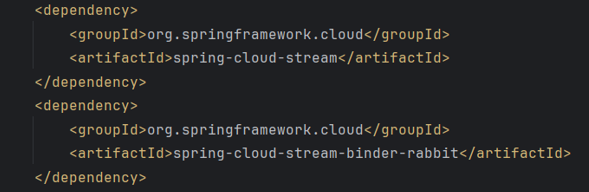
**Implementation**

**Part 1: Communication Between Accounts Microservice and Message Microservice Using RabbitMQ**

**Changes in Accounts Microservice**

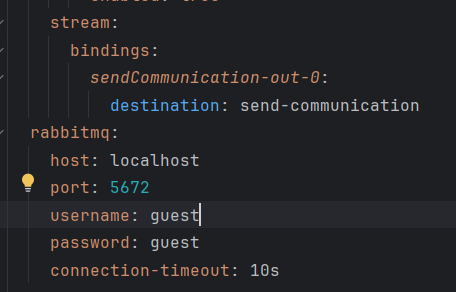
**Step 1: Add Required Dependencies to pom.xml**

Add the following dependencies for Spring Cloud Stream and RabbitMQ Binder:



**Step 2: Update application.yml File**

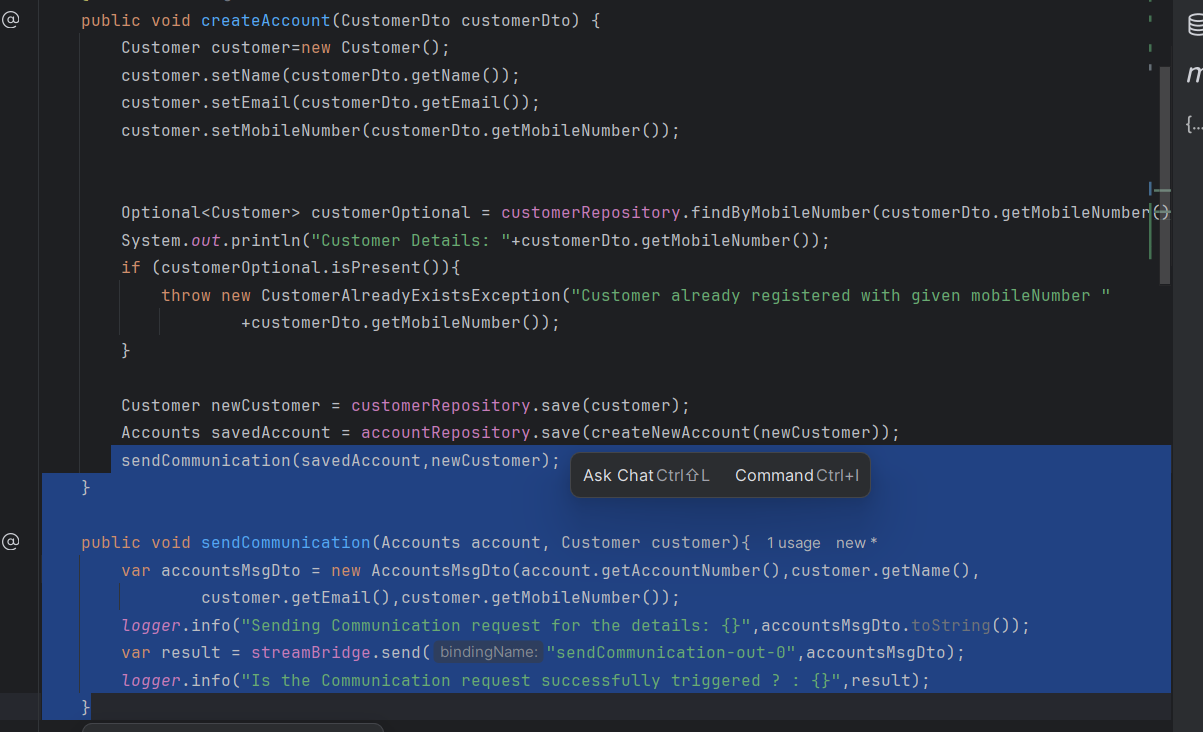
Specify the output channel and destination queue in the configuration:



* sendCommunication-out-0: Represents the **output channel**. The suffix -out-0 indicates it’s an output binding.
* destination: send-communication: Specifies the **queue name** where the event will be published.

**Step 3: Trigger Event in AccountsServiceImpl**

In the service implementation where the account is created, add the following snippet:

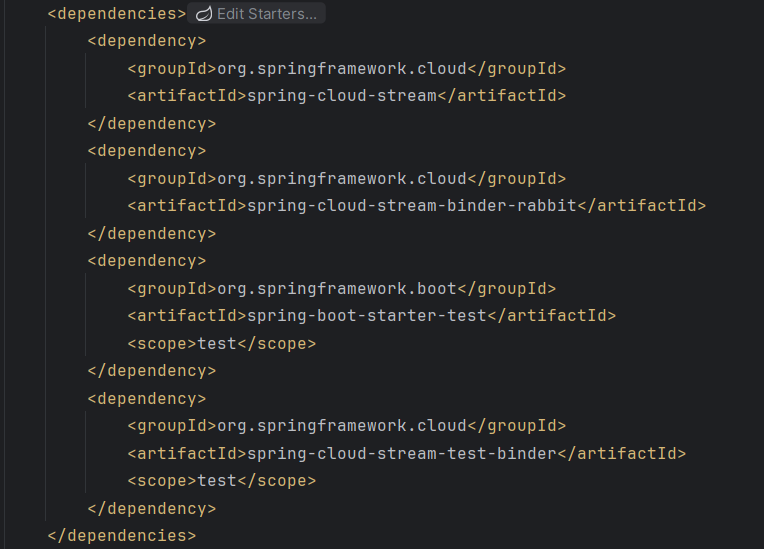


**Explanation**:

* Create a DTO (AccountsMsgDto) to pass relevant account details.
* Use the StreamBridge.send method to publish the event to RabbitMQ.

**Changes added in Message Microservice**

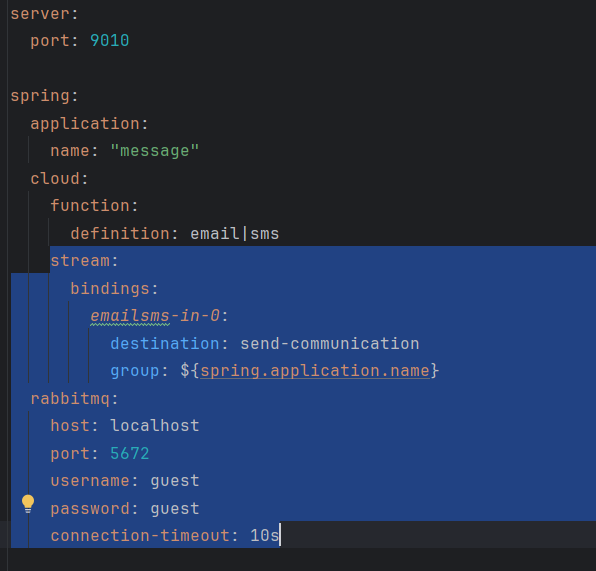
Step 1: Add Required Dependencies to pom.xml



**Note**: spring-cloud-function is **not required** because spring-cloud-stream already includes functional support.

**Step 2: Update application.yml File**

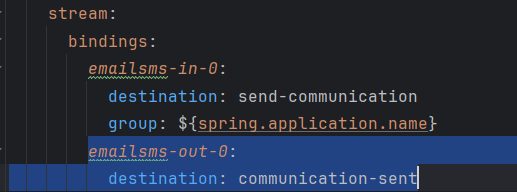
Configure the input channel to listen for events:



* emailsms-in-0: Represents the **input channel**. The suffix -in-0 indicates it’s an input binding.
* destination: send-communication: Specifies the **queue name** from which messages will be consumed.
* The event sent by the **Accounts Microservice** to the queue will be consumed by the **Message Microservice**. The Message Microservice will process the event and perform its respective tasks, such as sending **email** or **SMS** notifications, as defined in the spring.cloud.function.definition property.

**Part 2: Communication between Message Microservice to Accounts Microservice**

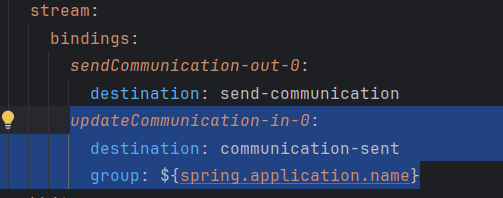
**Changes added in Message Microservice**

****

* emailsms-out-0: Represents the **output channel** to publish acknowledgment events.
* destination: communication-sent: Specifies the queue name where acknowledgment messages will be published.

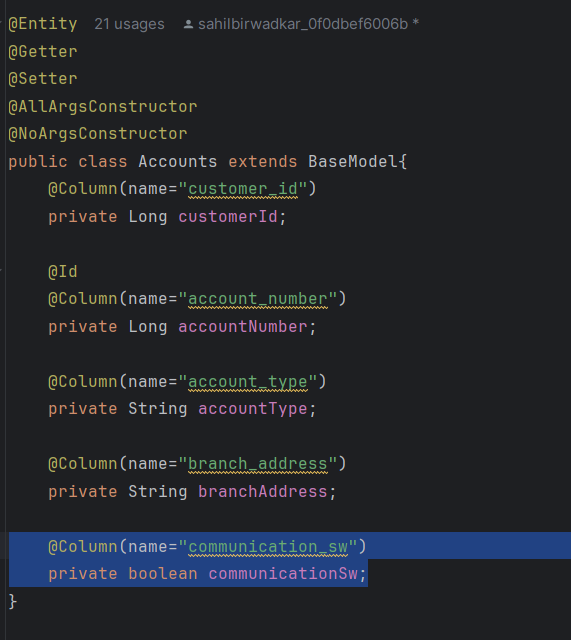
**Changes added in Accounts Microservice**

Configure the input channel to listen for acknowledgment events:



* updateCommunication-in-0: Represents the **input channel**.
* destination: communication-sent: Specifies the **queue name** from which acknowledgment messages will be consumed.

Once we get the account number, we’ve to make changes in our accounts db indicating we’ve sent email, sms to the customer and for that we’ve to add column under accounts table.



Also, in schema.sql file we’ll add same parameter

**Output Behavior**

Let's consider the following scenario:

* We place a debugging point (breakpoint) in the Message Microservice at a specific point in the code.
* At this point, the Message Microservice will pause execution and wait for further instructions (e.g., resume execution via debugging tools).

**Behavior in Accounts Microservice**

* Even though the **Message Microservice** is paused, the **Accounts Microservice** will **still return a successful response** to the end user confirming that the account has been created.
* This is because the **event publishing** from the Accounts Microservice to RabbitMQ is **asynchronous**. The Accounts Microservice does not wait for the Message Microservice to process the task.

**Behavior in Message Microservice**

* Once we resume execution in the **Message Microservice**, it will:
  1. **Consume the message** from the RabbitMQ queue (where the event is stored).
  2. Perform the required task, such as sending an **email** or **SMS** notification to the respective customer.

**Summary**

* The Accounts Microservice works **independently** and responds to the end user without being impacted by the state of the Message Microservice.
* RabbitMQ ensures that the **event (task)** remains in the queue until the Message Microservice resumes execution and processes it.
* This asynchronous behavior ensures reliability and decoupling between services, making the system more resilient.