TDMQ for Apache Pulsar

SDK Documentation

Product Documentation

Contents

SDK Documentation

SDK Overview

Last updated：2024-06-28 11:33:56

TDMQ for Apache Pulsar supports TCP protocol (Pulsar Community Edition) and HTTP protocol. The following are the supported multi-language SDKs:

Note:

In order to be more consistent with the Pulsar open-source community, we have stopped feature updates for the Tencent Cloud SDK since April 30, 2021. We recommend that you use the Apache Pulsar SDK for TDMQ for Apache Pulsar.

Protocol type	SDK language
TCP protocol (Pulsar Community Edition)	Go SDK
	Java SDK
	C++ SDK
	Python SDK
	Node.js SDK
HTTP protocol	Go SDK
	Java SDK
	C++ SDK
	Python SDK
	PHP SDK

Apache Pulsar TCP Protocol Spring Boot Starter

Last updated：2024-06-28 11:33:56

Overview

This document describes how to use Spring Boot Starter to send and receive messages and helps you better understand the message sending and receiving processes.

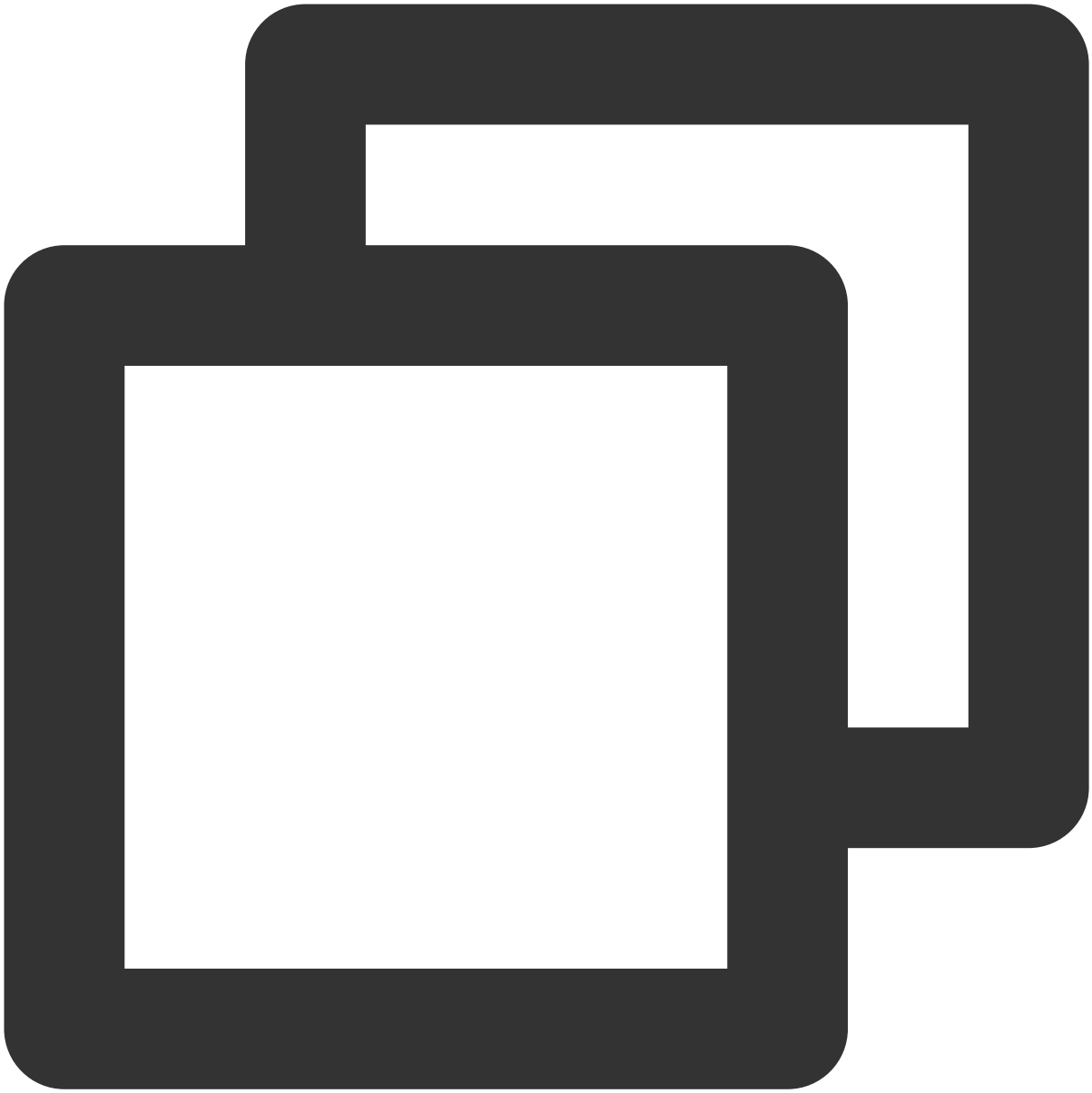
Prerequisites

You have created the required resources.
You have installed JDK 1.8 or later
You have installed Maven 2.5 or later
You have downloaded the demo

Directions

Step 1. Add dependencies

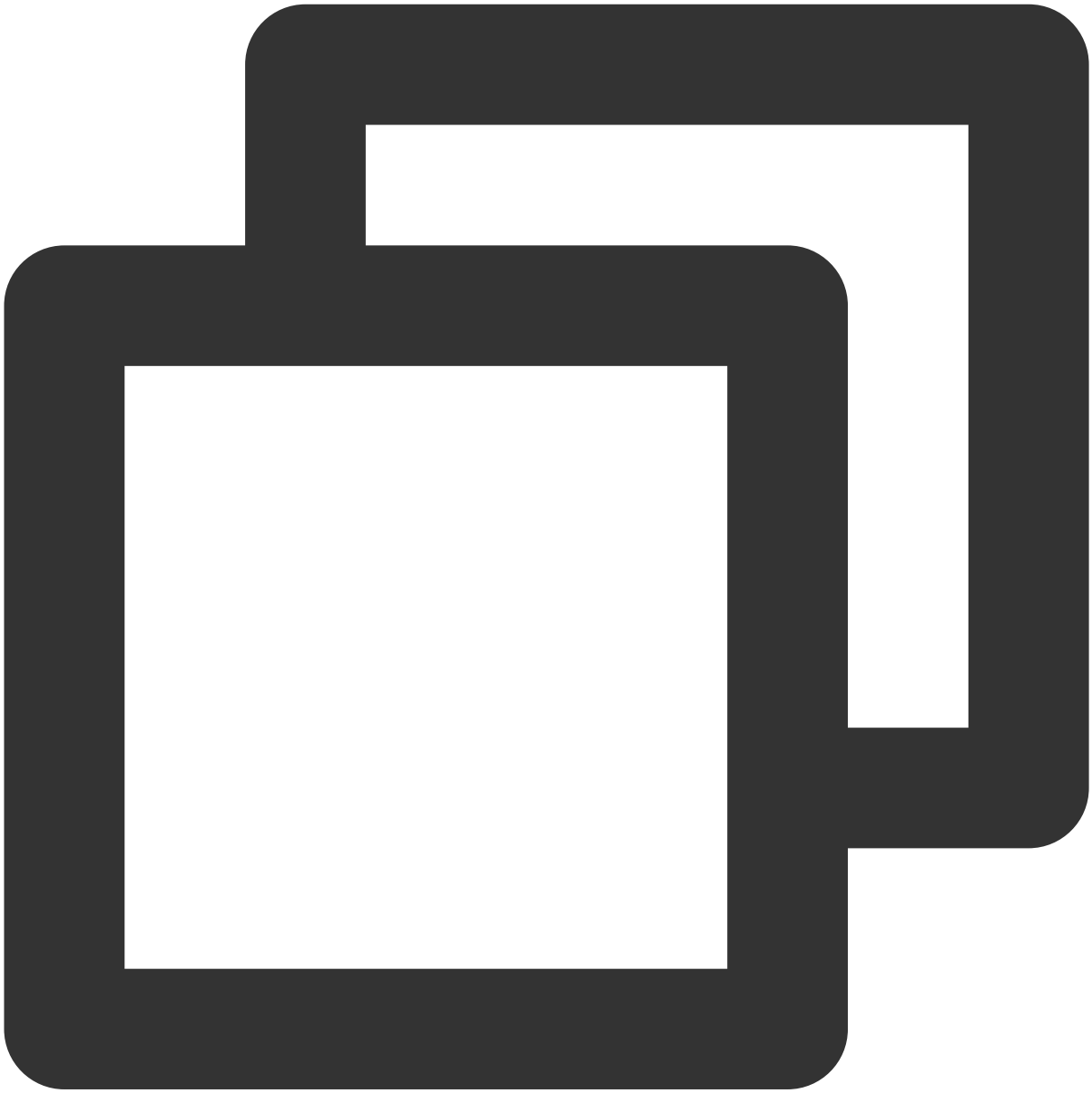
Import Pulsar Starter dependencies to the project.



```
<dependency>
    <groupId>io.github.majusko</groupId>
    <artifactId>pulsar-java-spring-boot-starter</artifactId>
    <version>1.0.7</version>
</dependency>
<!-- https://mvnrepository.com/artifact/io.projectreactor/reactor-core -->
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.4.11</version>
</dependency>
```

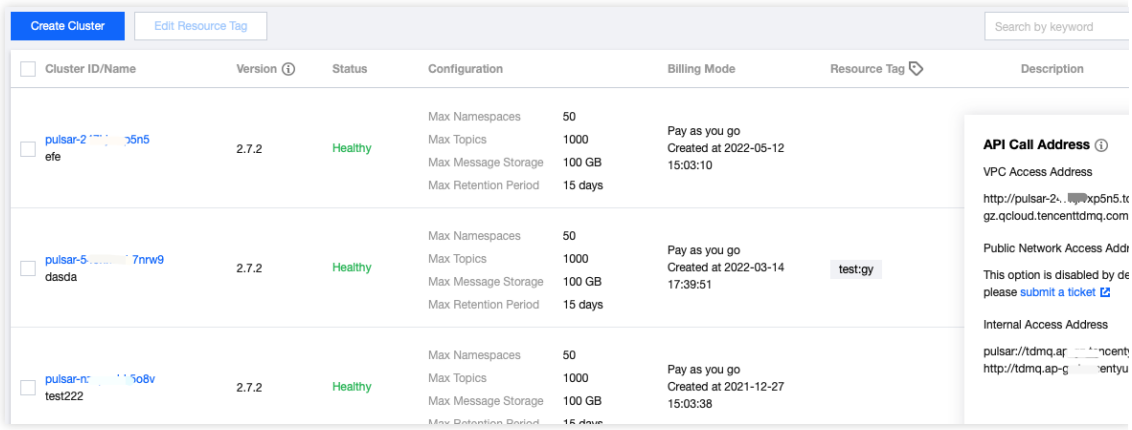
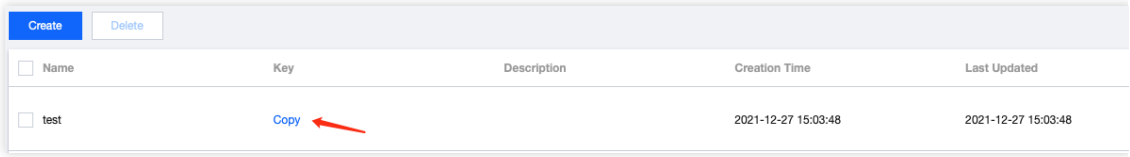
Step 2. Prepare configurations

Add Pulsar configuration information to the configuration file.



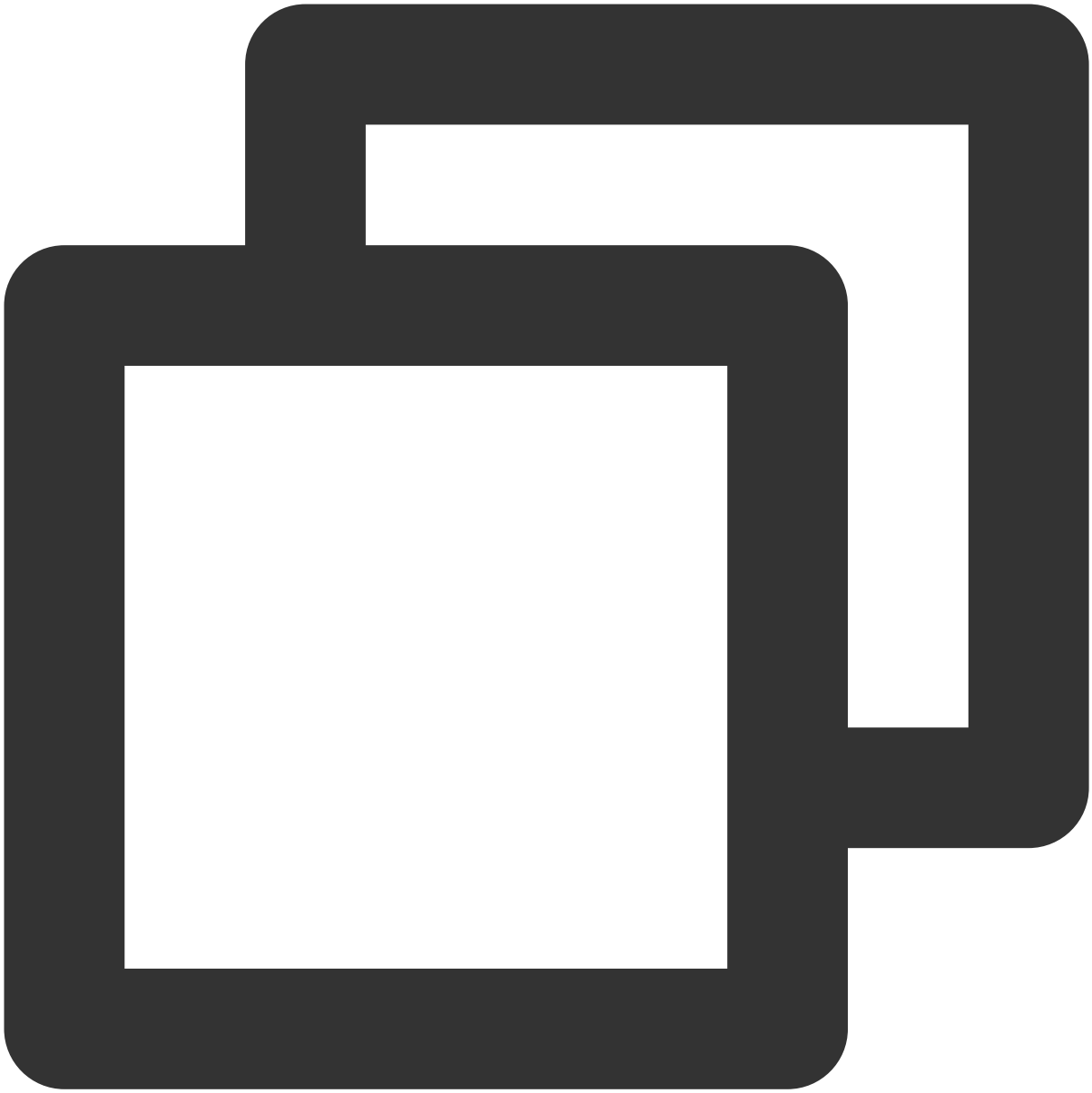
```
pulsar:
  # Namespace name
  namespace: namespace_java
  # Service access address
  service-url: http://pulsar-xxx.tdmq.ap-gz.public.tencenttdmq.com:8080
```

```
# Role token
token-auth-value: eyJrZXlJZC....
# Cluster name
tenant: pulsar-xxx
```

Parameter	Description
namespace	Namespace name, which can be copied on the Namespace page in the console.
service-url	Cluster access address, which can be viewed and copied on the Cluster page in the console. 
token-auth-value	Role token, which can be copied in the **Token** column on the Role Management page. 
tenant	Cluster ID, which can be obtained on the Cluster page in the console.

Step 3. Produce messages

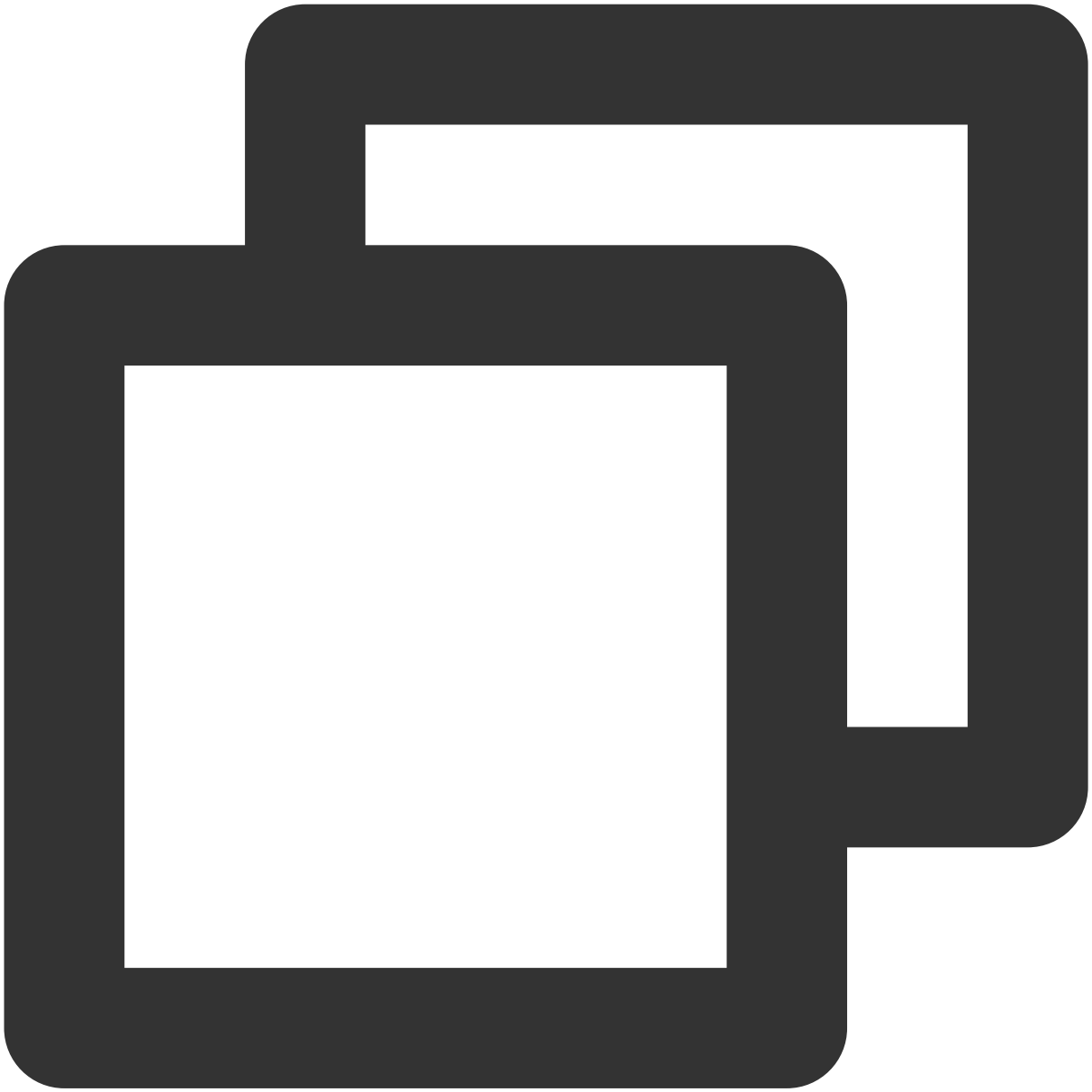
1. Configure the producer.



```
@Configuration
public class ProducerConfiguration {

    @Bean
    public ProducerFactory producerFactory() {
        return new ProducerFactory()
                // topic1
                .addProducer("topic1")
                // topic2
                .addProducer("topic2");
    }
```

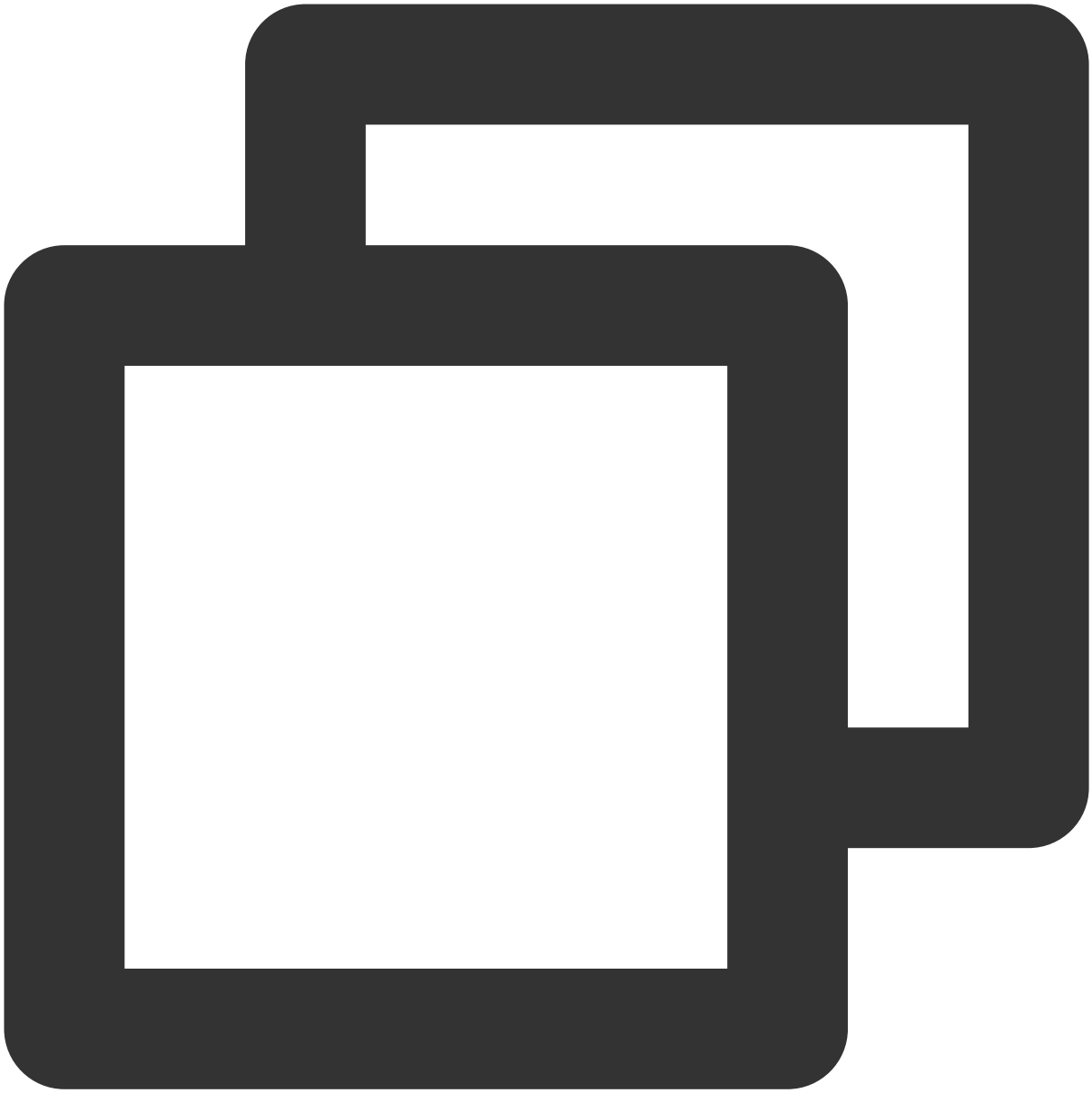
```
}
```

2. Inject the producer.



```
@Autowired
private PulsarTemplate<byte[]> defaultProducer;
```

3. Send messages.



```
// Send the messages
defaultProducer.send("topic2", ("Hello pulsar client, this is a order message.").ge
```

Note:

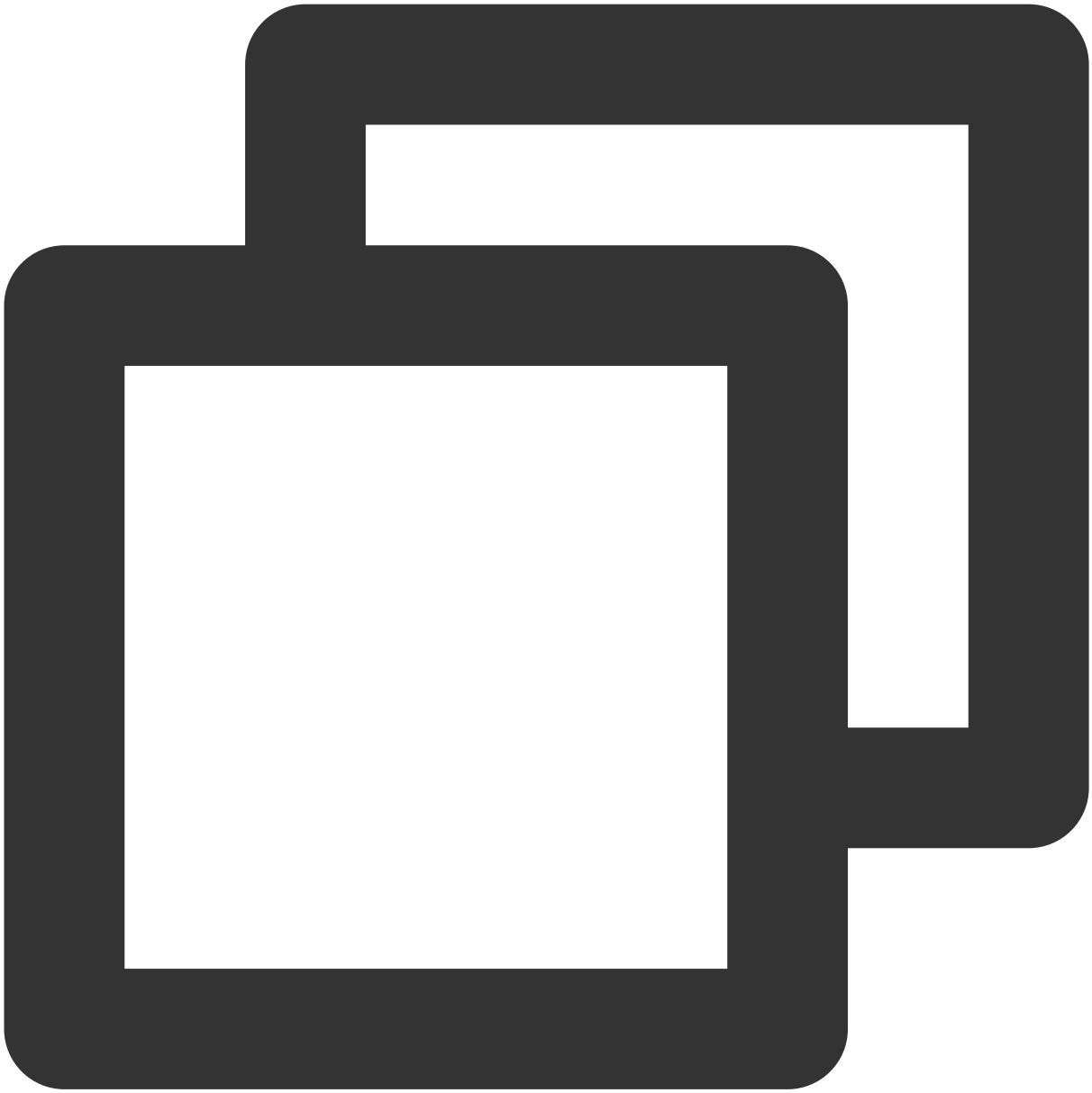
The topic that sends messages is the one declared in the producer configuration.

The type of `PulsarTemplate` must be the same as that of the sent message.

When you send a message to the specified topic, the message type must be the same as that bound to the topic in the producer factory configuration.

Step 4. Consume messages

Configure the consumer.

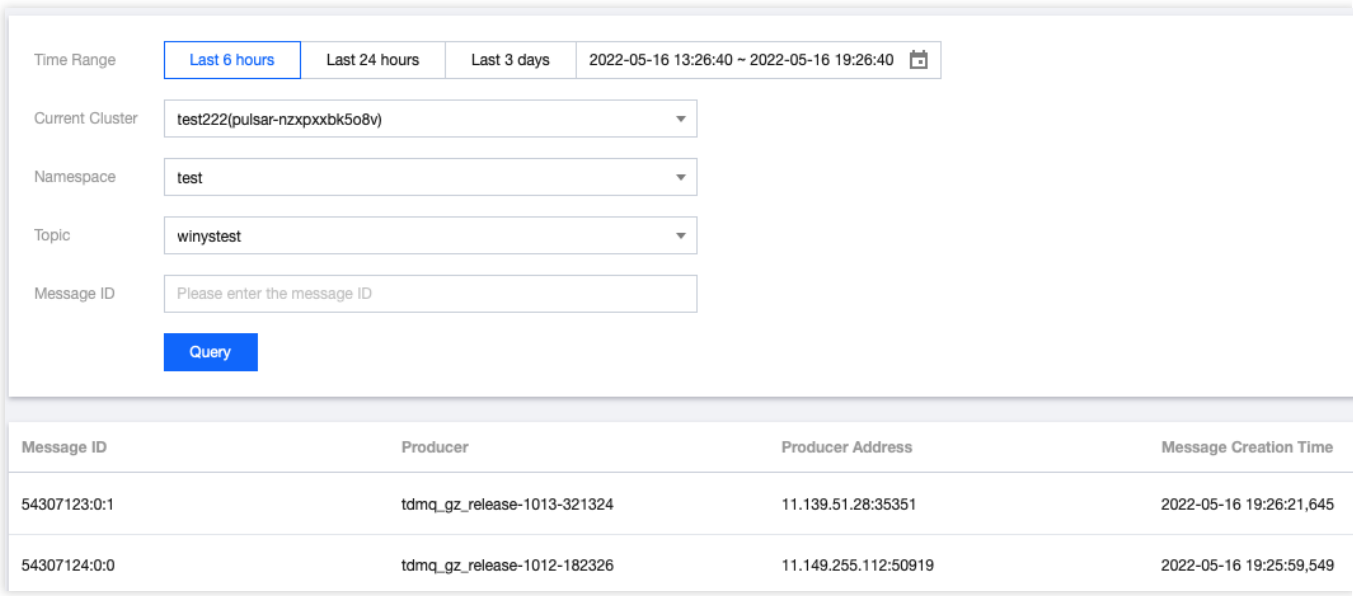


```
@PulsarConsumer(topic = "topic1",  // Name of the subscribed topic
                subscriptionName = "sub_topic1", // Subscription name
                serialization = Serialization.JSON, // Serialization method
                subscriptionType = SubscriptionType.Shared, // Subscription mode, w
                consumerName = "firstTopicConsumer", // Consumer name
                maxRedeliverCount = 3, // Maximum number of retries
                deadLetterTopic = "sub_topic1-DLQ" // Dead letter topic name
               )
```

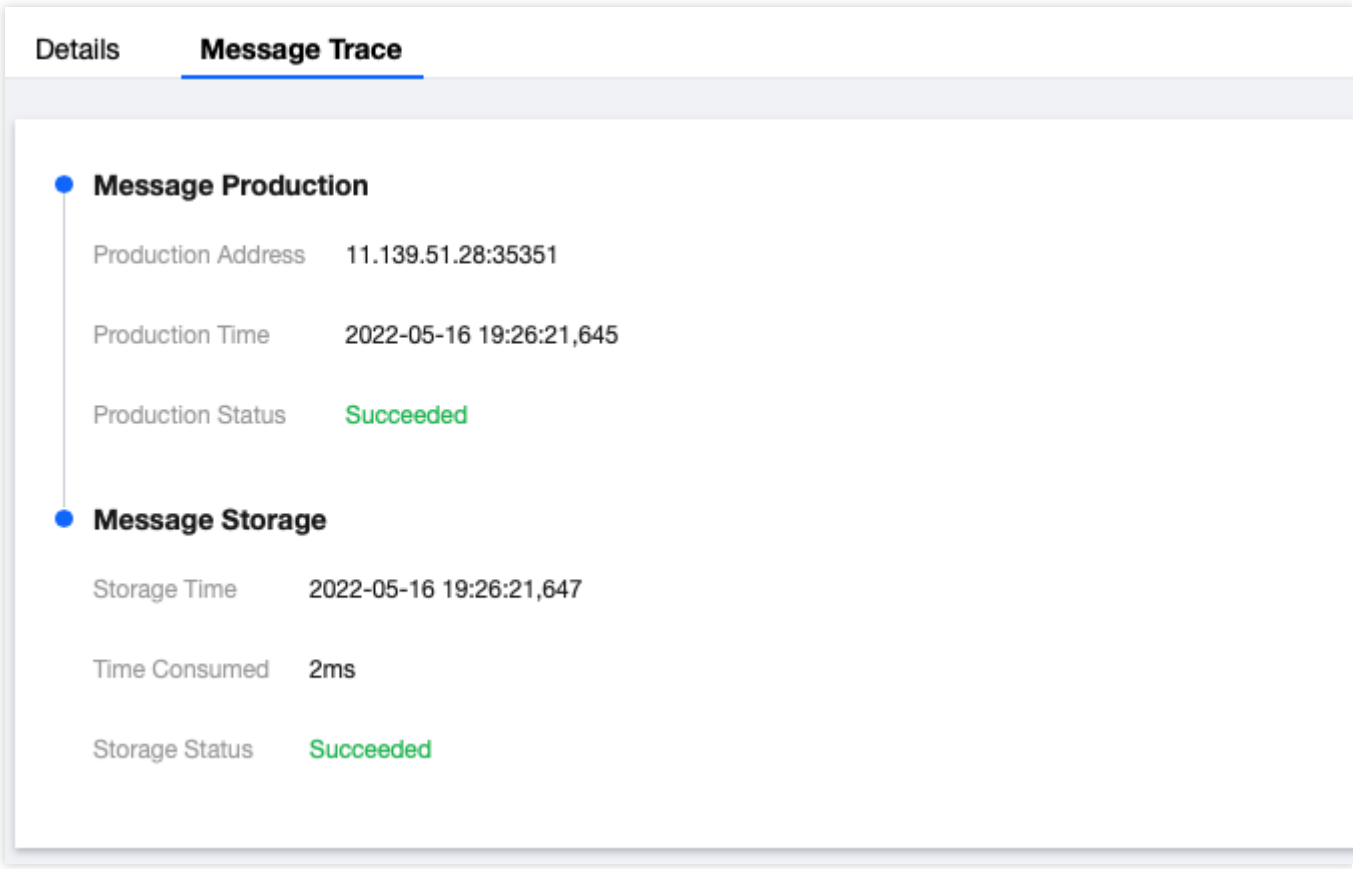
```
public void topicConsume(byte[] msg) {
    // TODO process your message
    System.out.println("Received a new message. content: [" + new String(msg) + "]"
    // If the consumption fails, throw an exception, so that the message will enter
}
```

Step 5. Query messages

Log in to the console and enter the **Message Query** page to view the message trace after running the demo.



The message trace is as follows:



Note:

The above is a simple configuration for using TDMQ for Apache Pulsar through Spring Boot Starter. For more information, see Demo or Spring Boot Starter for Apache Pulsar.

SDK for Java

Last updated：2024-06-28 11:33:56

Scenarios

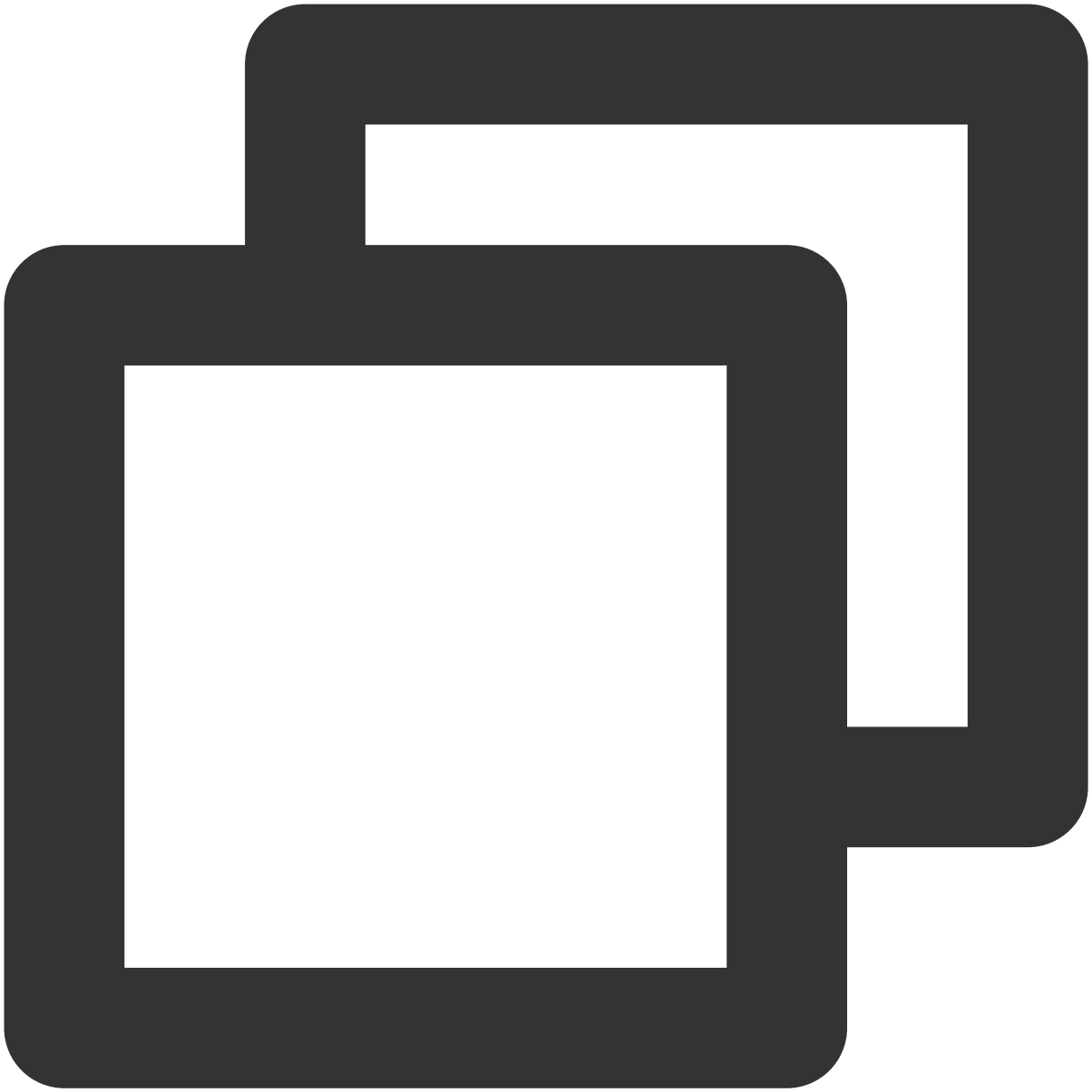
This document describes how to use open-source SDK to send and receive messages by using the SDK for Java as an example and helps you better understand the message sending and receiving processes.

Prerequisites

You have created the required resources.
You have installed JDK 1.8 or later
You have installed Maven 2.5 or later
You have downloaded the demo

Directions

1. Introduce dependencies in a Java project and add the following dependencies to the `pom.xml` file. This document uses a Maven project as an example.



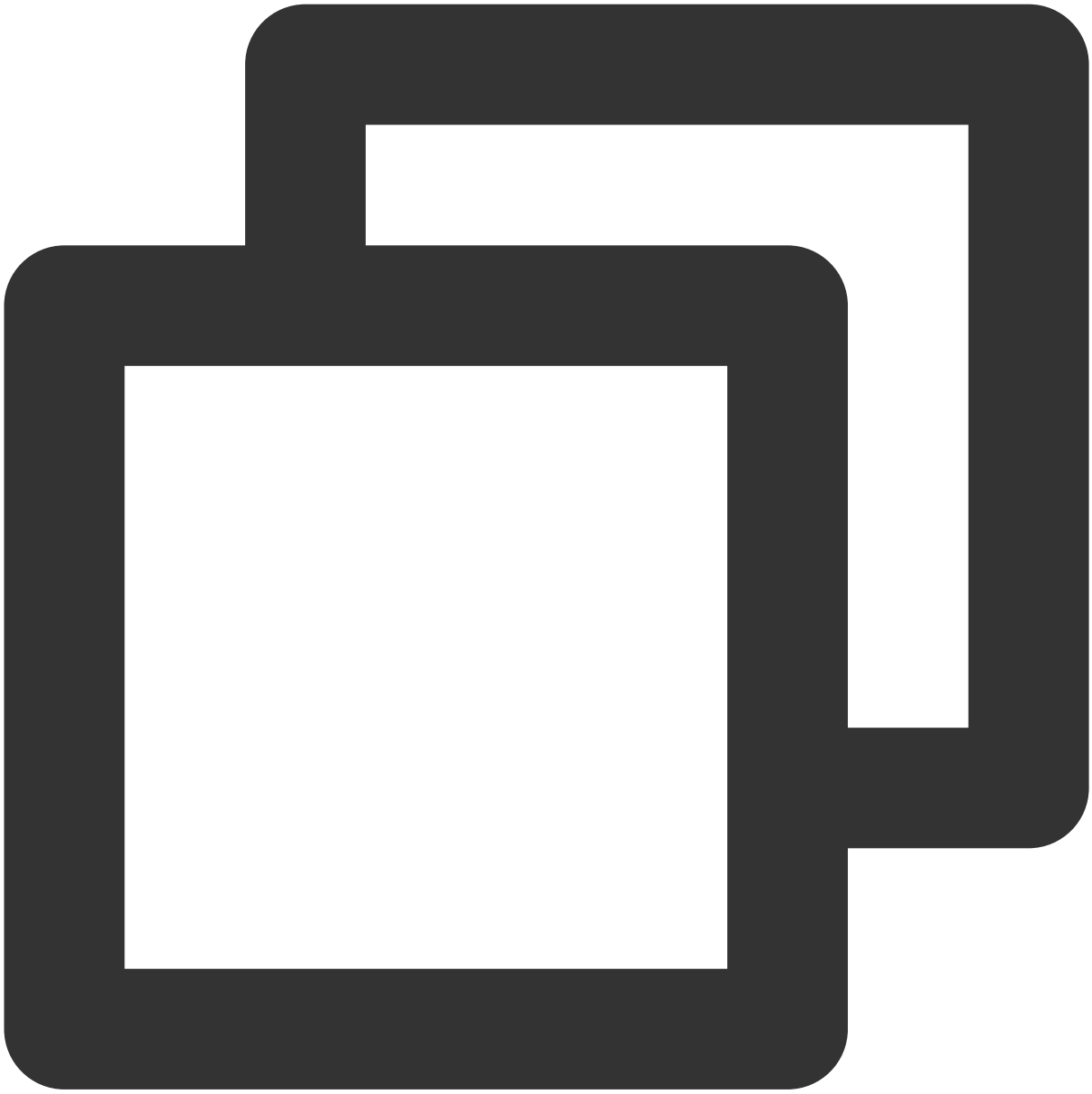
```
<dependency>
        <groupId>org.apache.pulsar</groupId>
        <artifactId>pulsar-client</artifactId>
        <version>2.7.2</version>
</dependency>
```

Note:

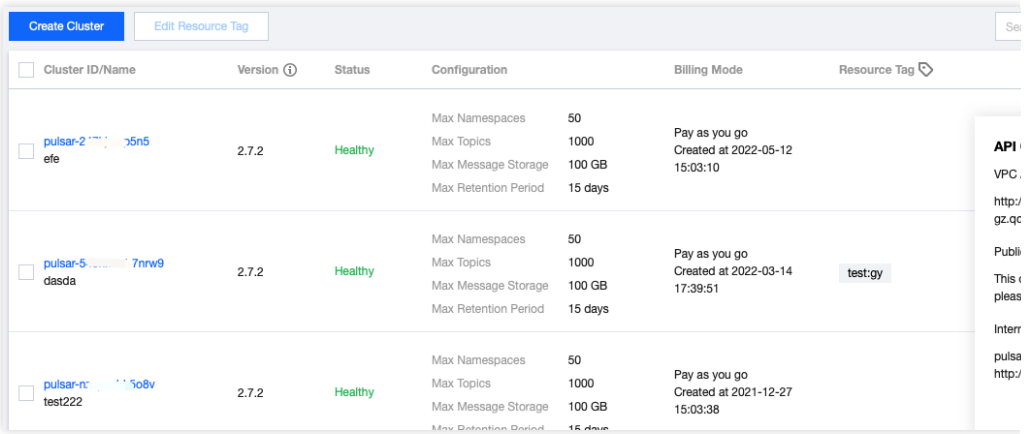
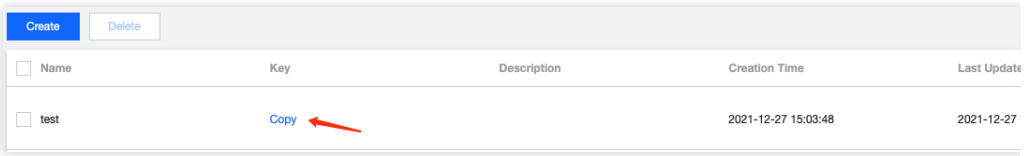
SDK 2.7.2 or later is recommended.

SDK 2.7.4 or later is recommended if you use the batch message sending and receiving feature (`BatchReceive`) of the client.

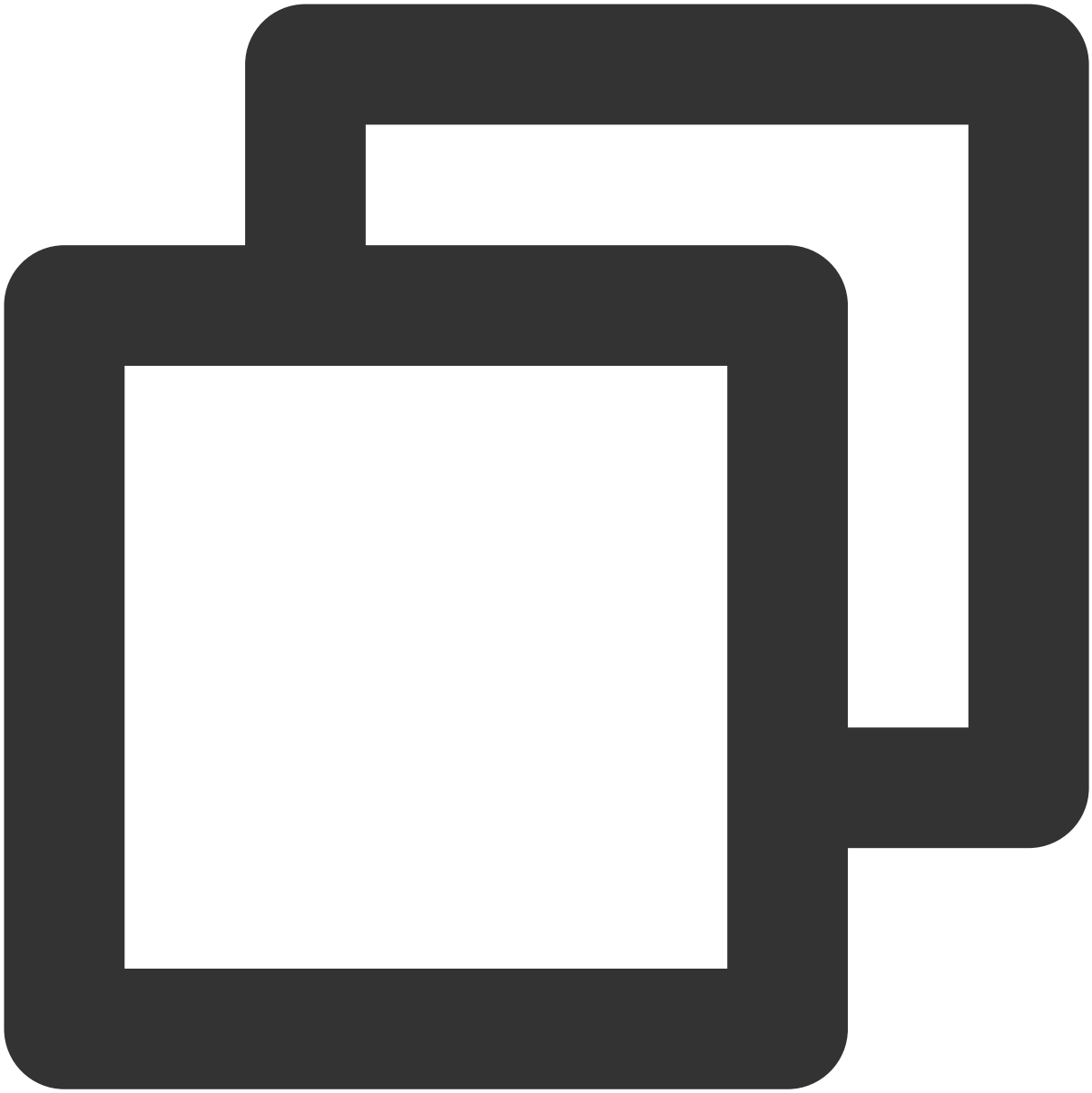
2. Create a Pulsar client.



```
PulsarClient pulsarClient = PulsarClient.builder()
                // Service access address
                .serviceUrl(SERVICE_URL)
                // Role token
                .authentication(AuthenticationFactory.token(AUTHENTICATION)).bui
```

Parameter	Description
SERVICE_URL	Cluster access address, which can be viewed and copied on the Cluster page in the consol 
AUTHENTICATION	Role token, which can be copied in the **Token** column on the Role Management page. 

3. Create a producer.

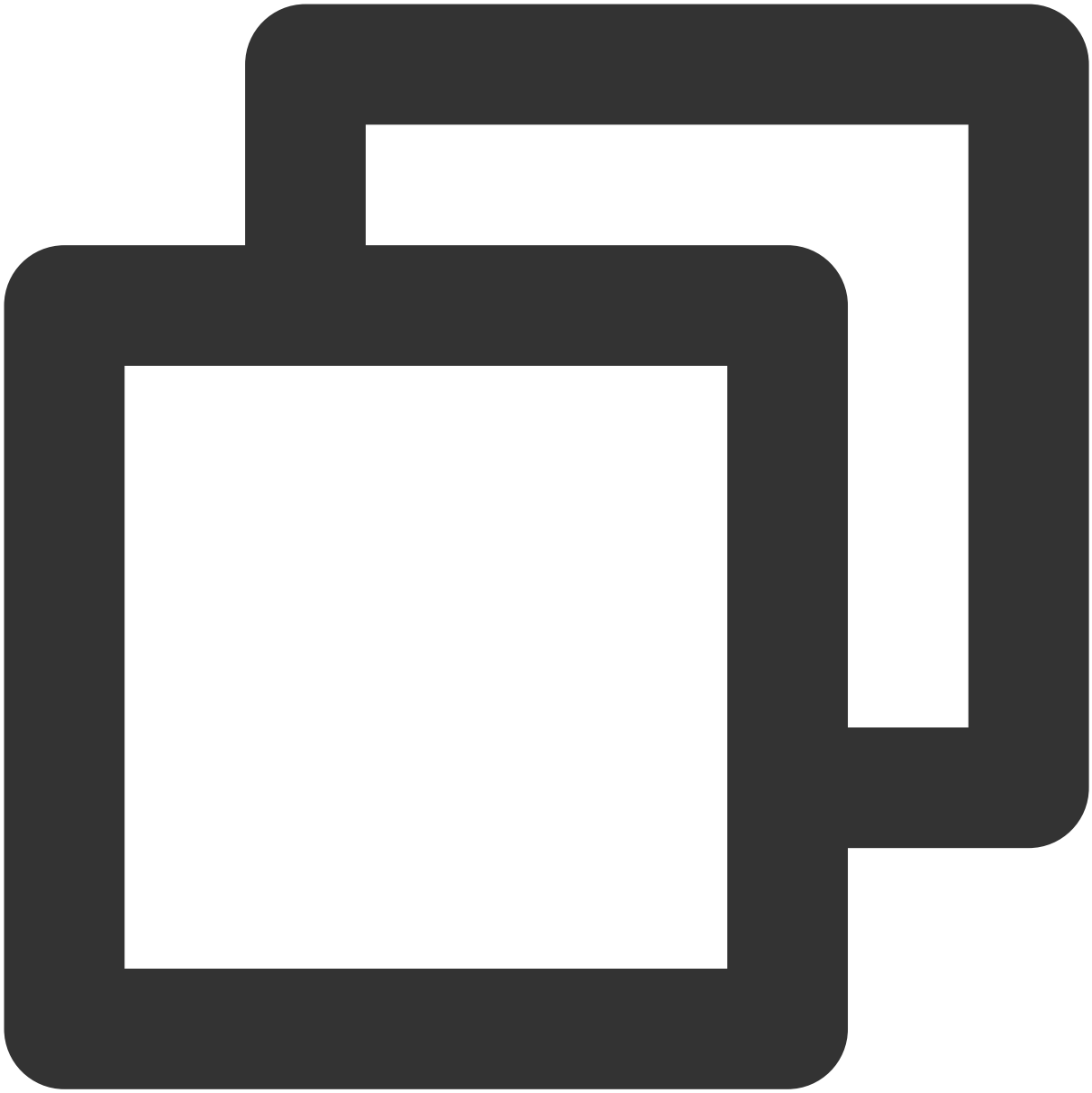


```
// Create a producer of the `byte[]` type
Producer<byte[]> producer = pulsarClient.newProducer()
    // Complete path of the topic in the format of `persistent://cluster (tenant
    .topic("persistent://pulsar-xxx/sdk_java/topic1").create();
```

Note:

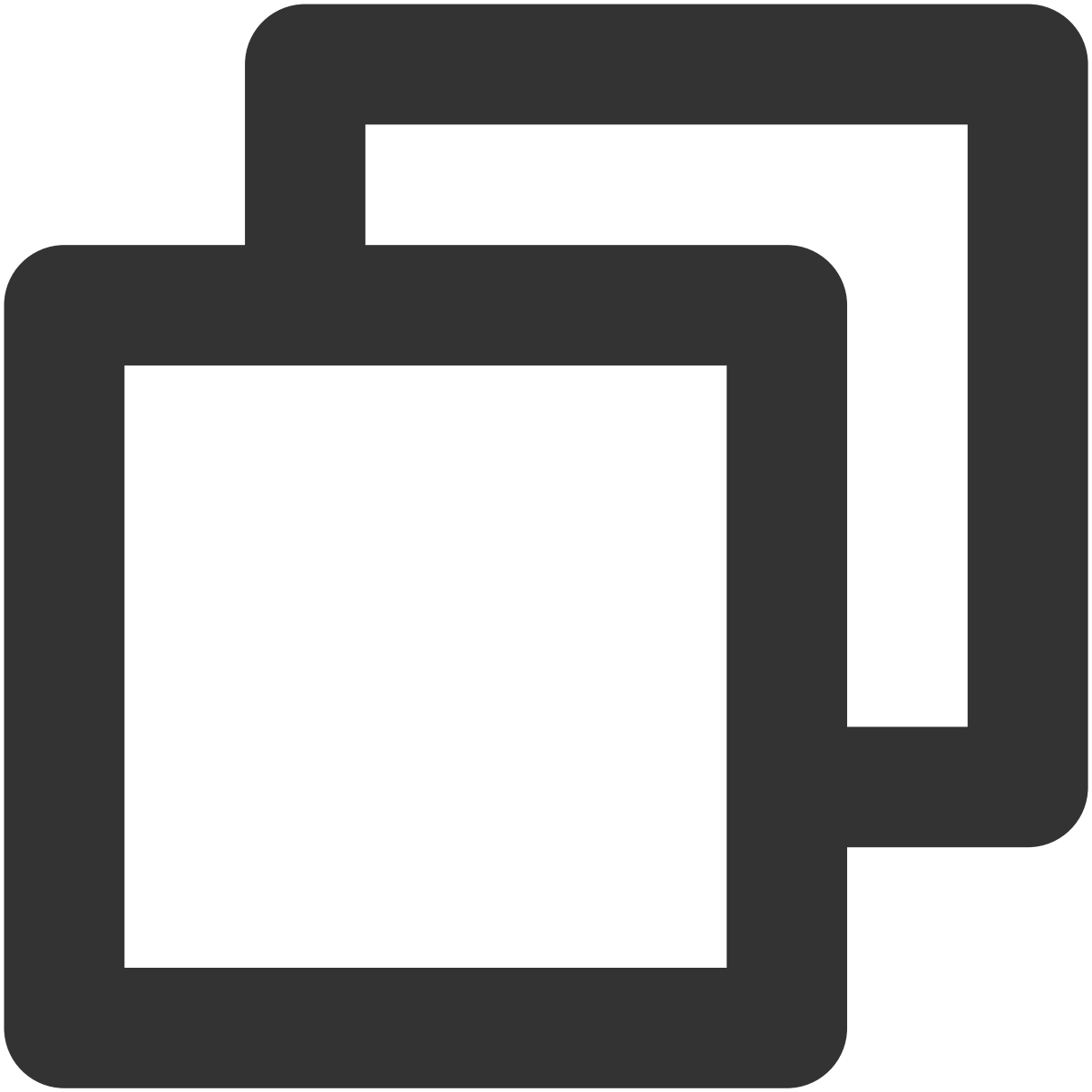
You need to enter the complete path of the topic name, i.e., `persistent://clusterid/namespace/Topic`, where the `clusterid/namespace/topic` part can be copied directly from the **Topic** page in the console.

4. Send the message.



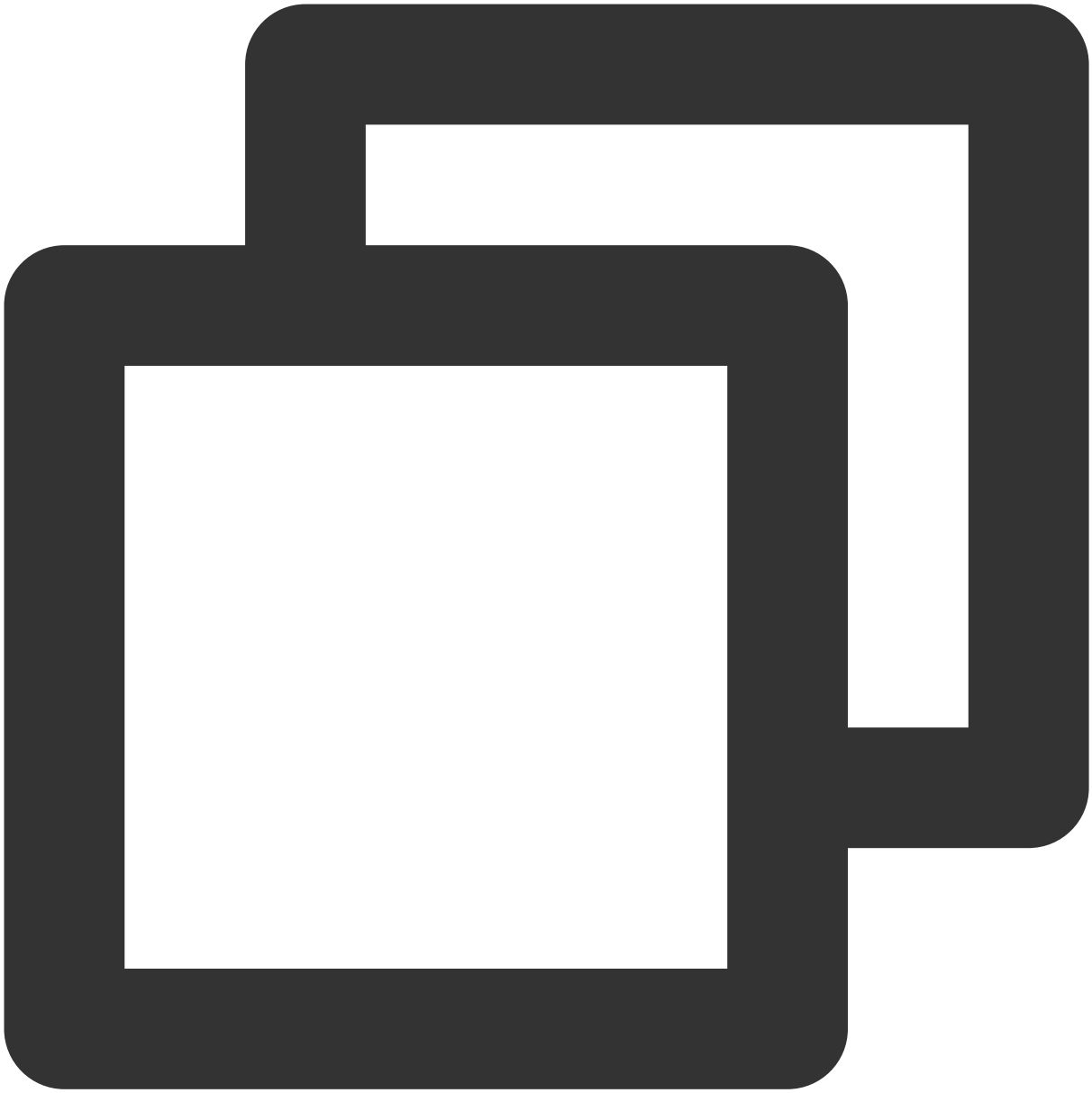
```
// Send the message
MessageId msgId = producer.newMessage()
    // Message content
    .value("this is a new message.".getBytes(StandardCharsets.UTF_8))
    // Business key
    .key("youKey")
    // Business parameter
    .property("mykey", "myvalue").send();
```

5. Release the resources.



```
// Disable the producer
producer.close();
// Disable the client
pulsarClient.close();
```

6. Create a consumer.

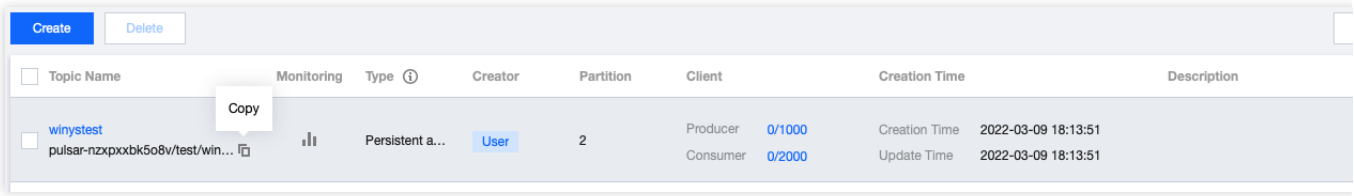


```
// Create a consumer of the `byte[]` type (default type)
Consumer<byte[]> consumer = pulsarClient.newConsumer()
    // Complete path of the topic in the format of `persistent://cluster (tenant
    .topic("persistent://pulsar-xxx/sdk_java/topic1")
    // You need to create a subscription on the topic details page in the consol
    .subscriptionName("sub_topic1")
    // Declare the exclusive mode as the consumption mode
    .subscriptionType(SubscriptionType.Exclusive)
    // Configure consumption starting at the earliest offset; otherwise, histori
    .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest)
    // Subscription
```

```
        .subscribe();
```

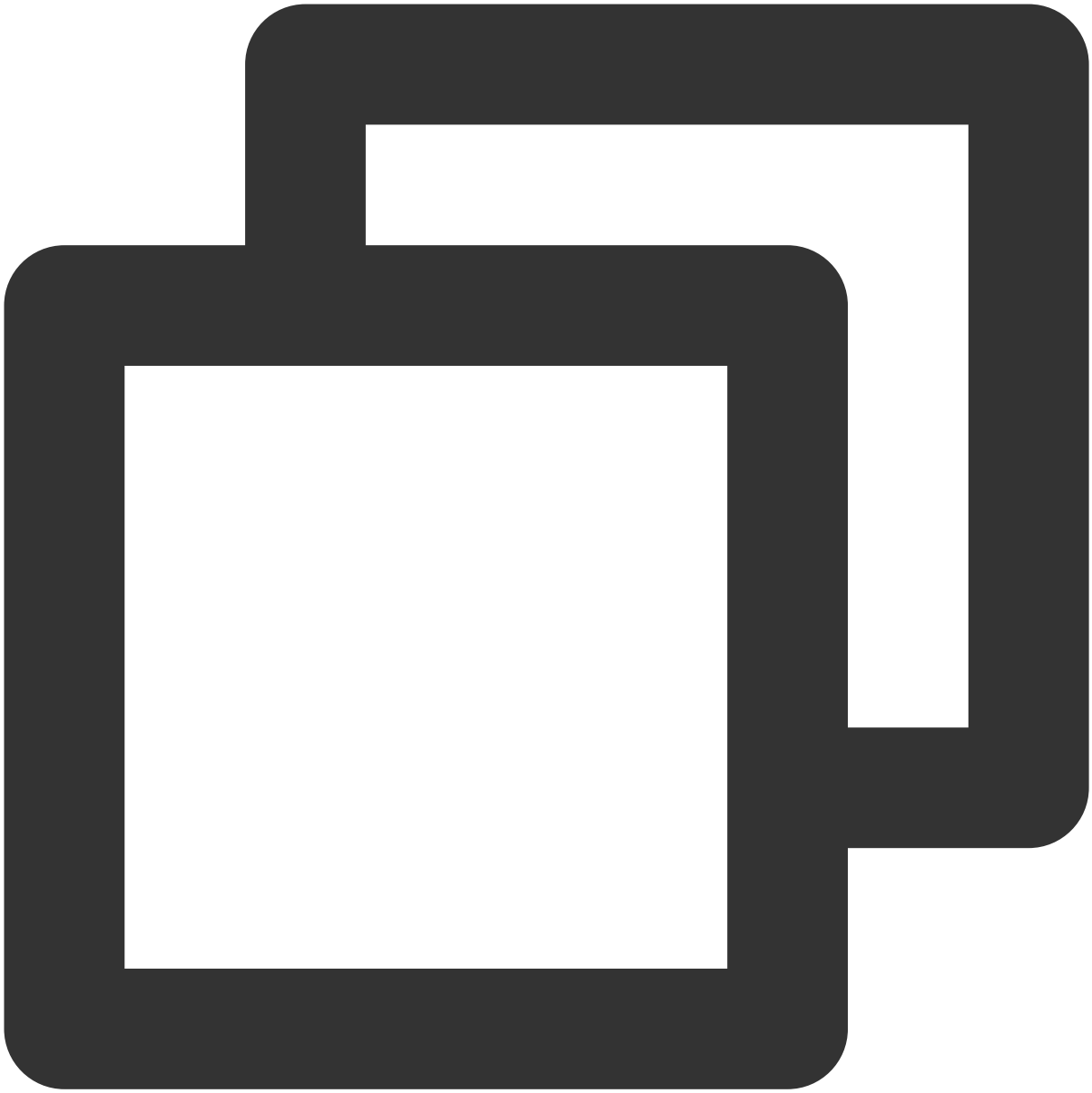
Note:

You need to enter the complete path of the topic name, i.e., `persistent://clusterid/namespace/Topic`, where the `clusterid/namespace/topic` part can be copied directly from the **Topic** page in the console.



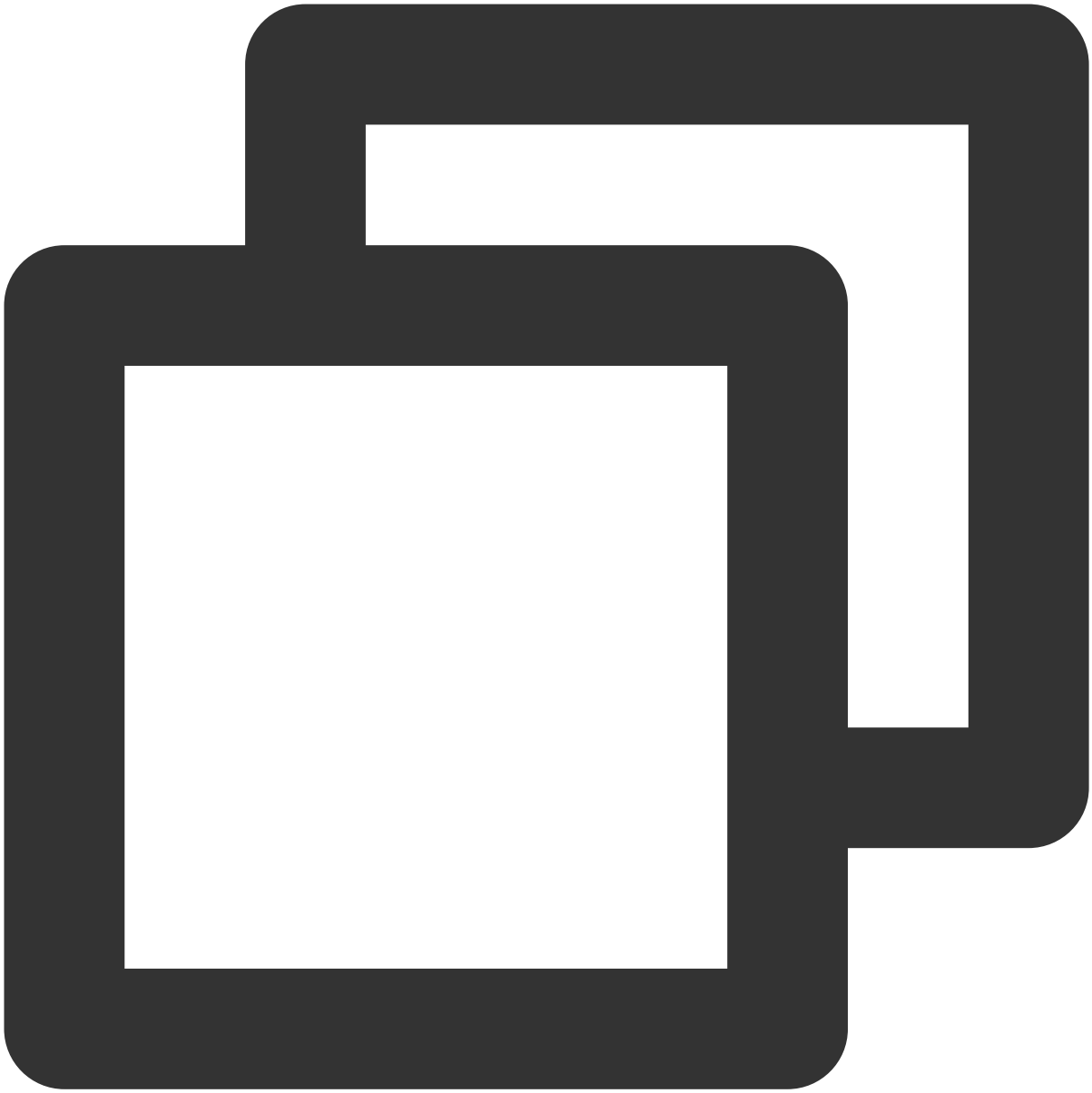
You need to enter the subscription name in the `subscriptionName` parameter, which can be viewed on the **Consumption Management** page.

7. Consume the message.



```
// Receive a message corresponding to the current offset
Message<byte[]> msg = consumer.receive();
MessageId msgId = msg.getMessageId();
String value = new String(msg.getValue());
System.out.println("receive msg " + msgId + ",value:" + value);
// Messages must be acknowledged after being received; otherwise, the offset wil
consumer.acknowledge(msg);
```

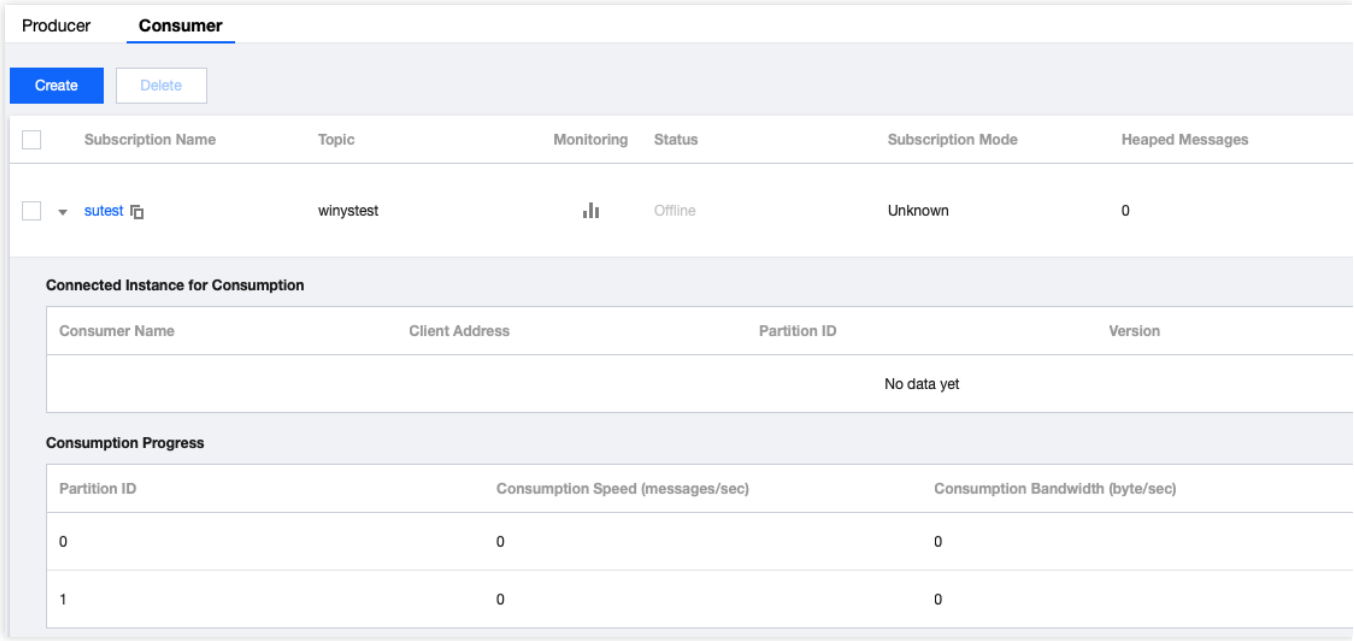
8. Use the listener for consumption.



```
// Message listener
MessageListener<byte[]> myMessageListener = (consumer, msg) -> {
    try {
        System.out.println("Message received: " + new String(msg.getData()));
        // Return `ack` as the acknowledgement
        consumer.acknowledge(msg);
    } catch (Exception e){
        // Return `nack` if the consumption fails
        consumer.negativeAcknowledge(msg);
    }
};
```

```
    pulsarClient.newConsumer()
        // Complete path of the topic in the format of `persistent://cluster (tenant
        .topic("persistent://pulsar-mmqwr5xx9n7g/sdk_java/topic1")
        // You need to create a subscription on the topic details page in the consol
        .subscriptionName("sub_topic1")
        // Declare the exclusive mode as the consumption mode
        .subscriptionType(SubscriptionType.Exclusive)
        // Set the listener
        .messageListener(myMessageListener)
        // Configure consumption starting at the earliest offset; otherwise, histori
        .subscriptionInitialPosition(SubscriptionInitialPosition.Earliest)
        .subscribe();
```

9. Log in to the TDMQ for Apache Pulsar console, click **Topic** > **Topic Name** to enter the **Consumption Management** page, and click the triangle below a subscription name to view the production and consumption records.



Note:

The above is a brief introduction to the way of publishing and subscribing to messages. For more operations, see Demo or Pulsar Java client.

SDK for Go

Last updated：2024-06-28 11:33:56

Overview

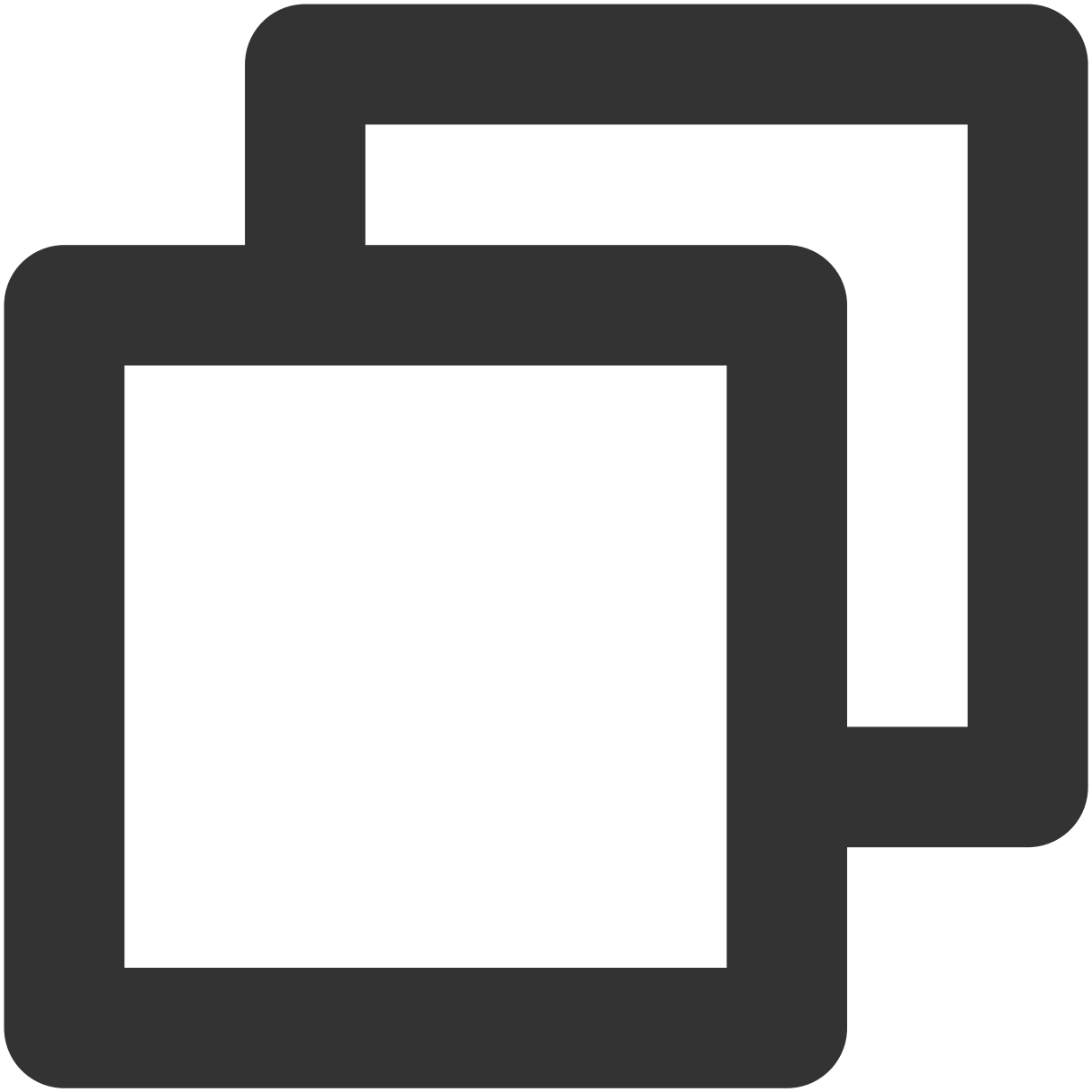
This document describes how to use open-source SDK to send and receive messages by using the SDK for Go as an example and helps you better understand the message sending and receiving processes.

Prerequisites

You have created the required resources as instructed in Resource Creation and Preparation.
You have installed Go.
You have downloaded the demo at here.

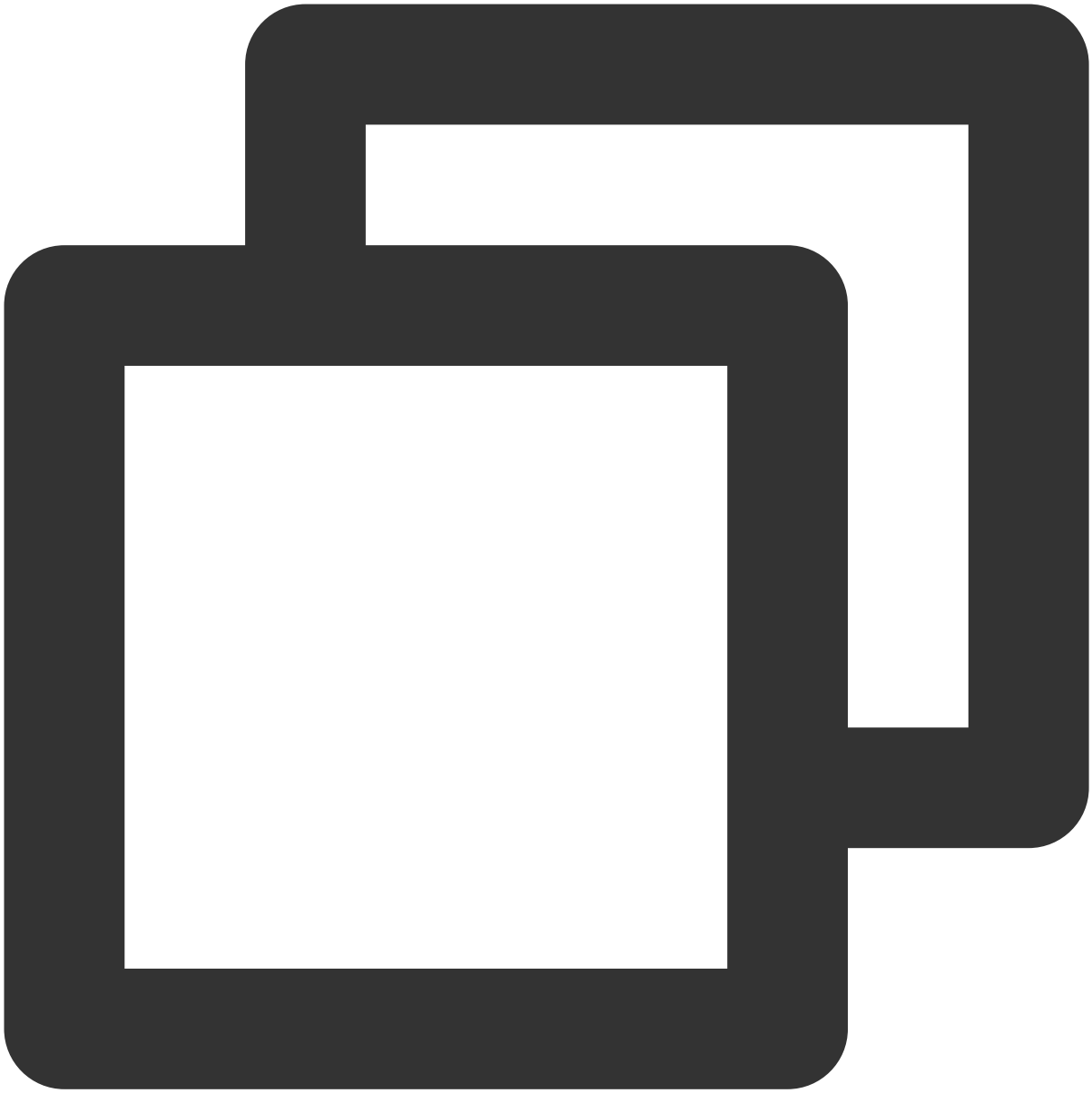
Directions

1. Import the `pulsar-client-go` library in the client environment.
1.1 Run the following command in the client environment to download the dependency package of the Pulsar client.



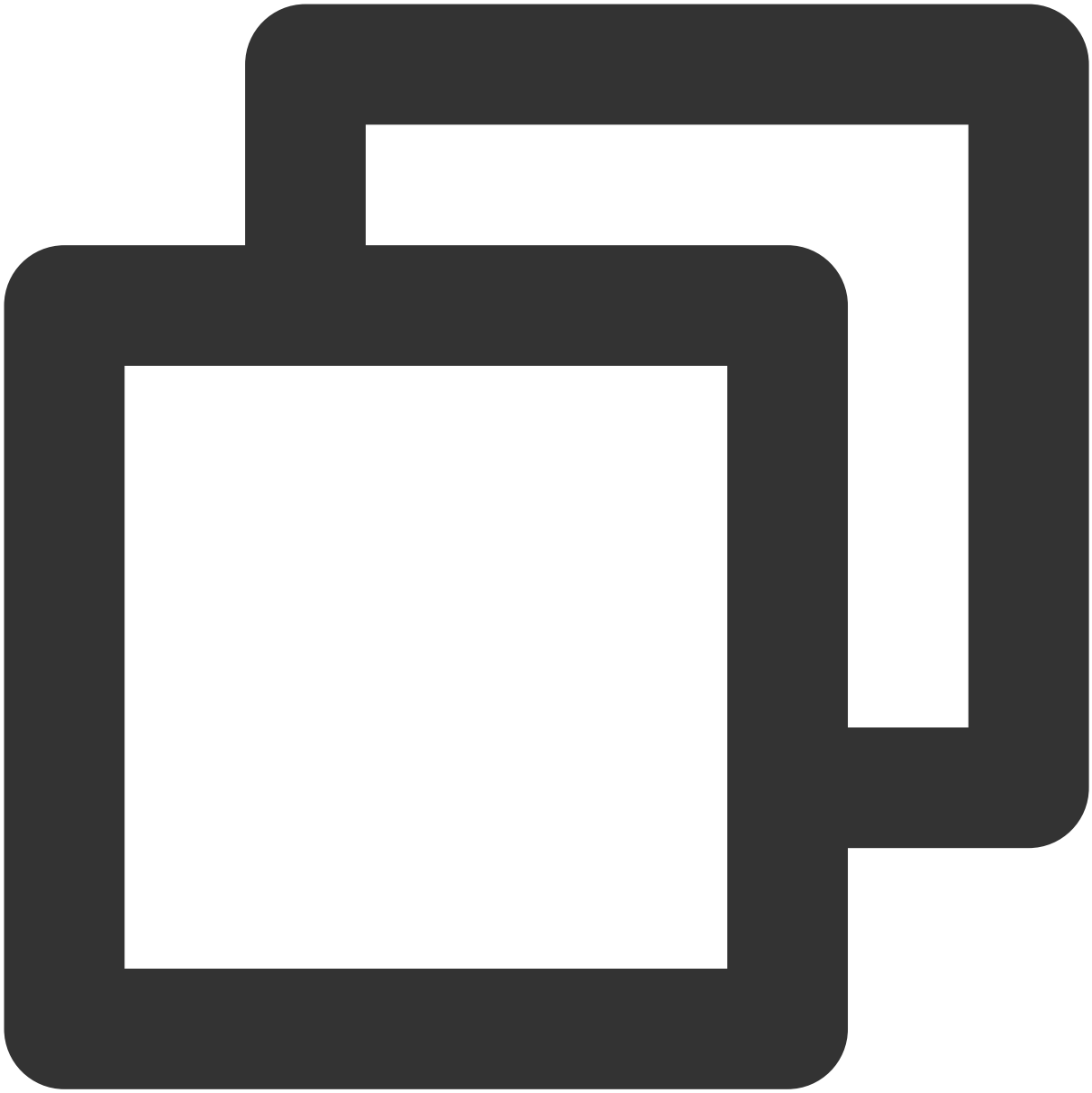
```
go get -u "github.com/apache/pulsar-client-go/pulsar"
```

1.2 After the installation is completed, use the following code to import the client into your Go project file.



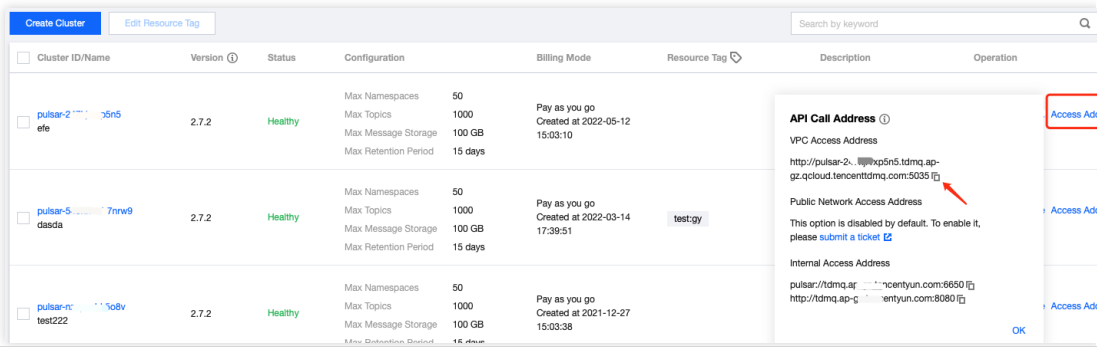
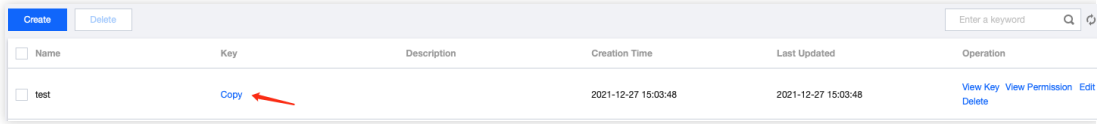
```
import "github.com/apache/pulsar-client-go/pulsar"
```

2. Create a Pulsar client.

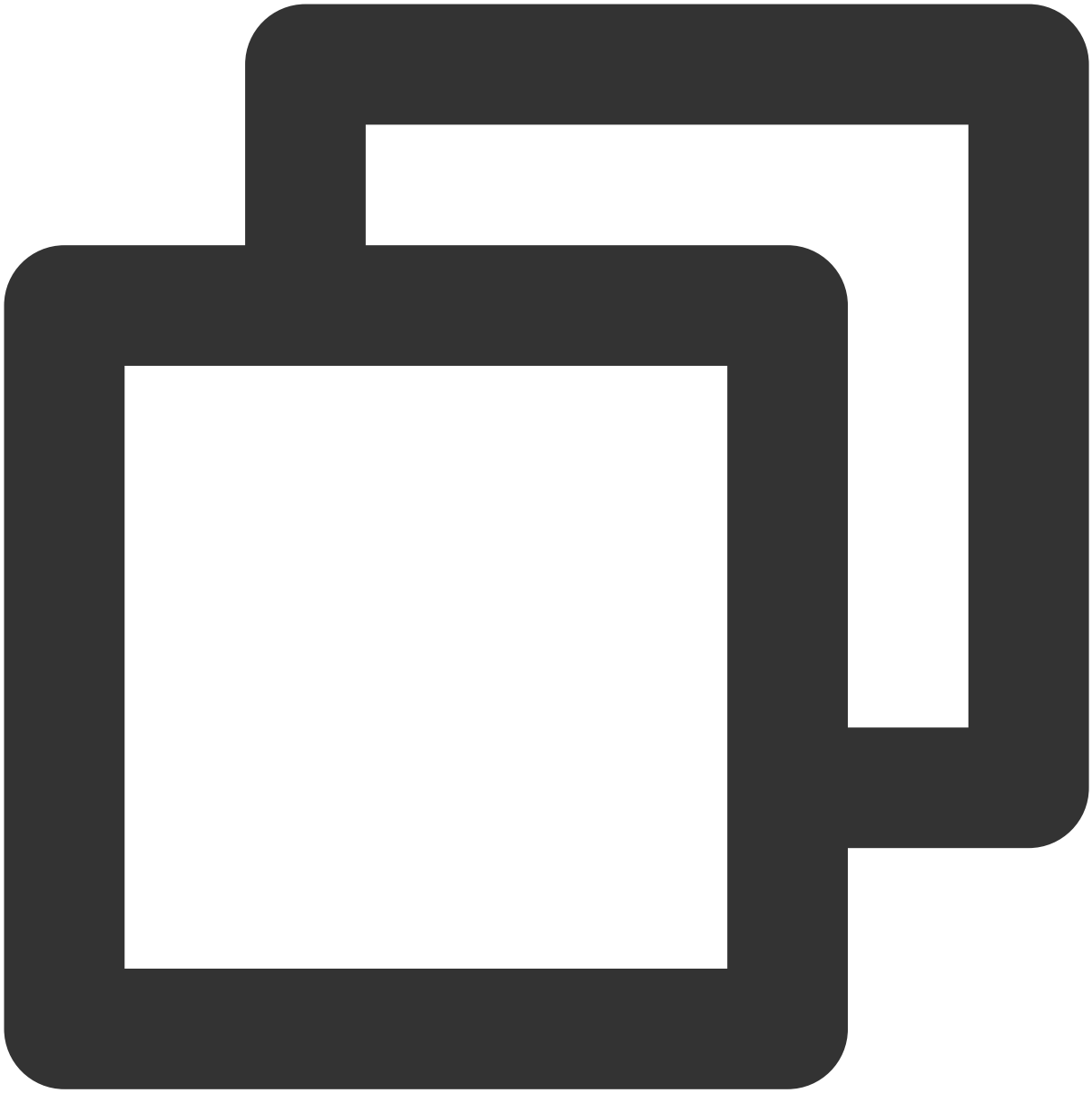


```
// Create a Pulsar client
client, err := pulsar.NewClient(pulsar.ClientOptions{
    // Service access address
    URL: serviceUrl,
    // Authorize the role token
    Authentication:    pulsar.NewAuthenticationToken(authentication),
    OperationTimeout:  30 * time.Second,
    ConnectionTimeout: 30 * time.Second,
})

if err != nil{
```

```
        log.Fatalf("Could not instantiate Pulsar client: %v", err)
    }

    defer client.Close()
```

Parameter	Description
serviceUrl	Cluster access address, which can be viewed and copied on the Cluster Management page in the 
Authentication	Role token, which can be copied in the **Token** column on the Role Management page. 

3. Create a producer.

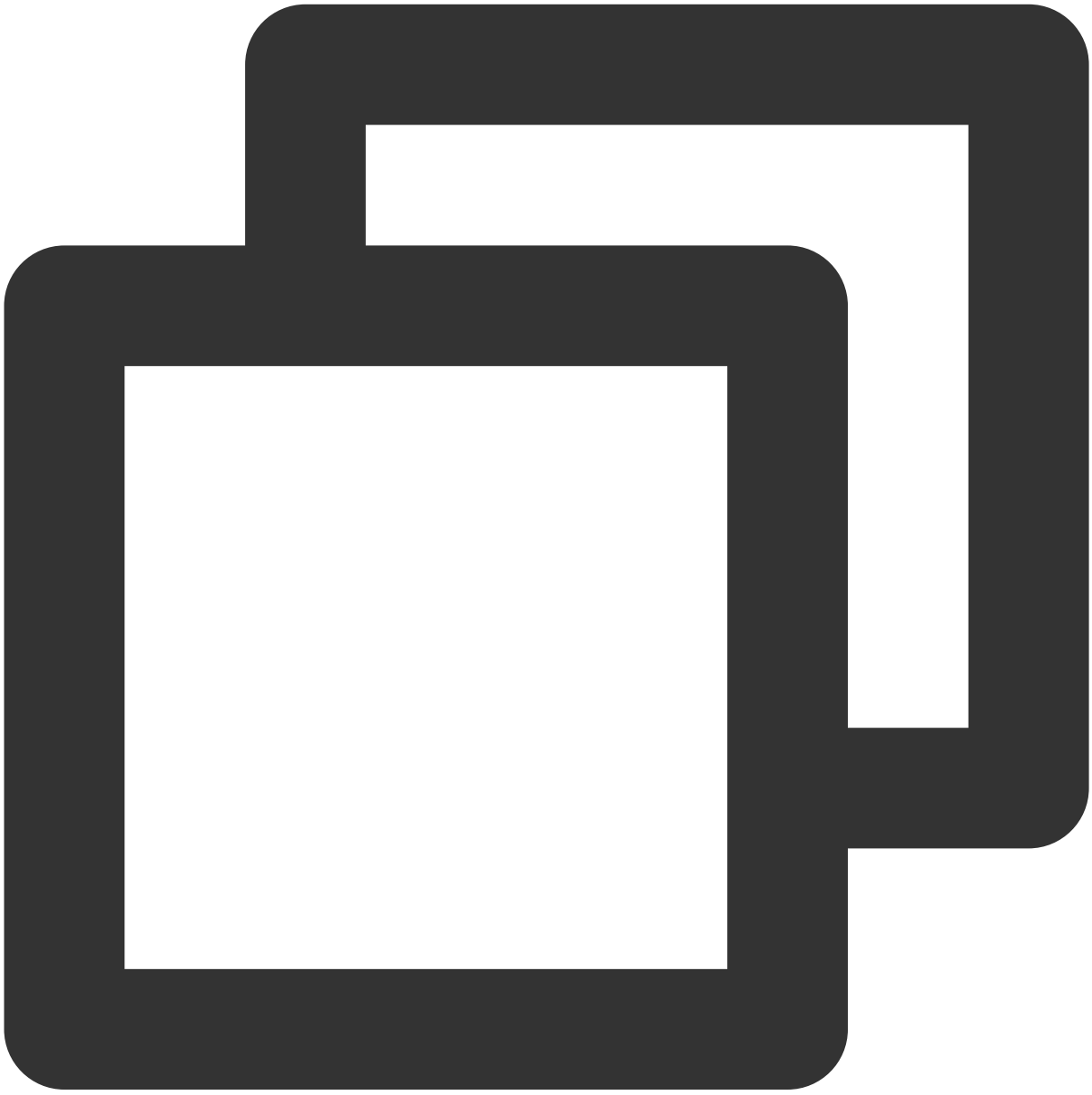


```
// Create a producer with the client
producer, err := client.CreateProducer(pulsar.ProducerOptions{
    // Complete path of the topic in the format of `persistent://cluster (tenant)
    Topic: "persistent://pulsar-mmqwr5xx9n7g/sdk_go/topic1",
})

if err != nil{
    log.Fatal(err)
}
defer producer.Close()
```

Note:

You need to enter the complete path of the topic name, i.e., `persistent://clusterid/namespace/Topic`, where the `clusterid/namespace/topic` part can be copied directly from the Topic Management page in the console.

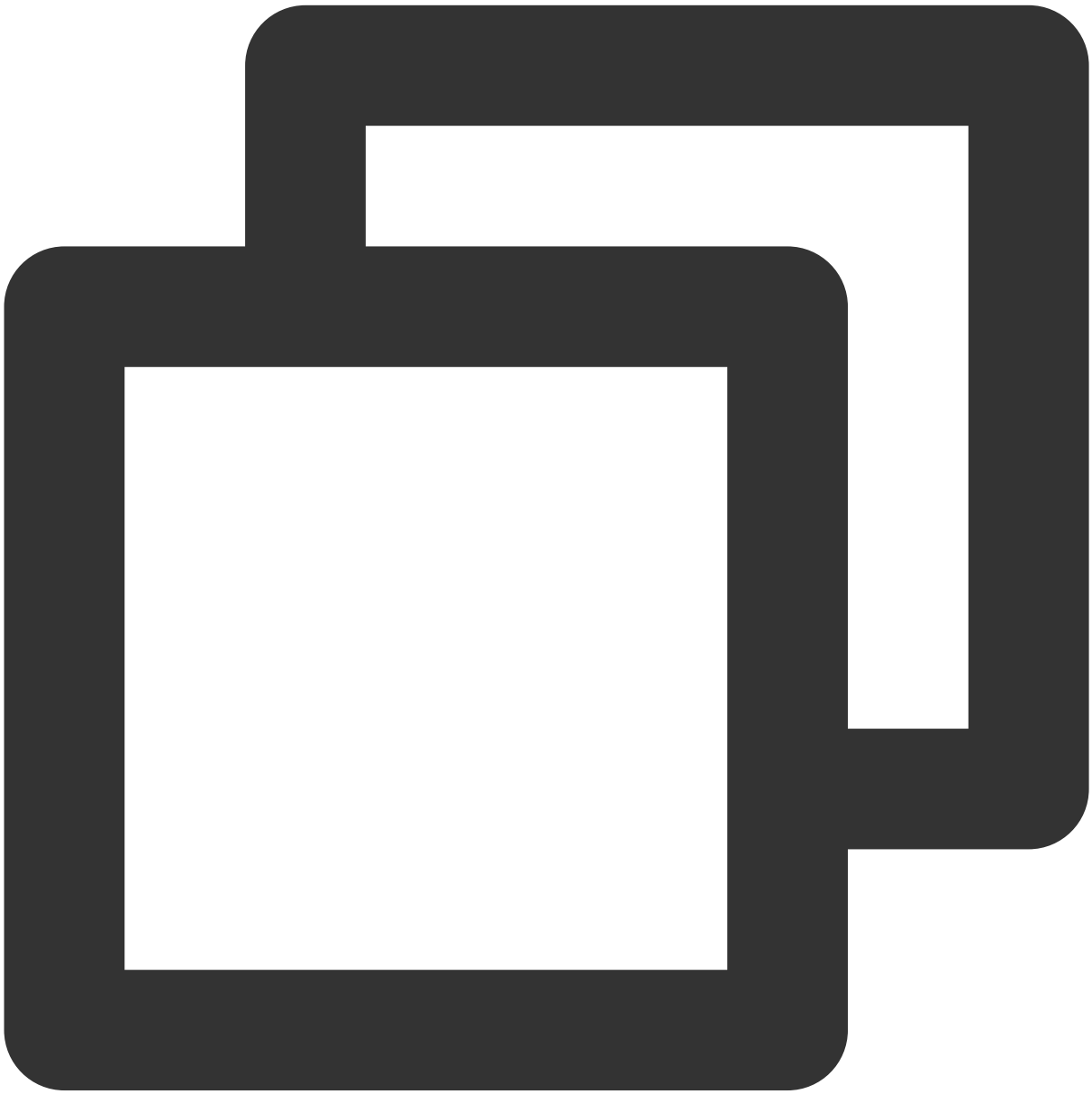
4. Send a message.



```
// Send the message
_, err = producer.Send(context.Background(), &pulsar.ProducerMessage{
    // Message content
    Payload: []byte("hello go client, this is a message."),
```

```
    // Business key
    Key: "yourKey",
    // Business parameter
    Properties: map[string]string{"key": "value"},
})
```

5. Create a consumer.

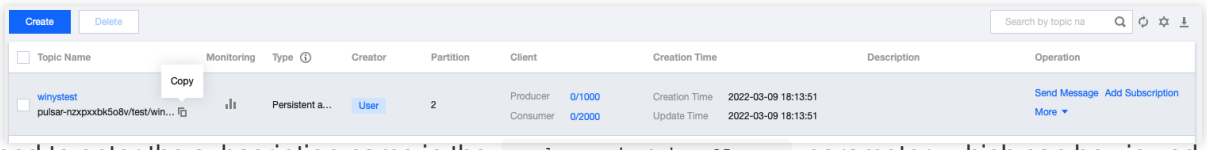


```
// Create a consumer with the client
consumer, err := client.Subscribe(pulsar.ConsumerOptions{
    // Complete path of the topic in the format of `persistent://cluster (tenant)
    Topic:            "persistent://pulsar-mmqwr5xx9n7g/sdk_go/topic1",
```

```
        // Subscription name
        SubscriptionName: "topic1_sub",
        // Subscription mode
        Type:              pulsar.Shared,
    })
    if err != nil{
        log.Fatal(err)
    }
    defer consumer.Close()
```

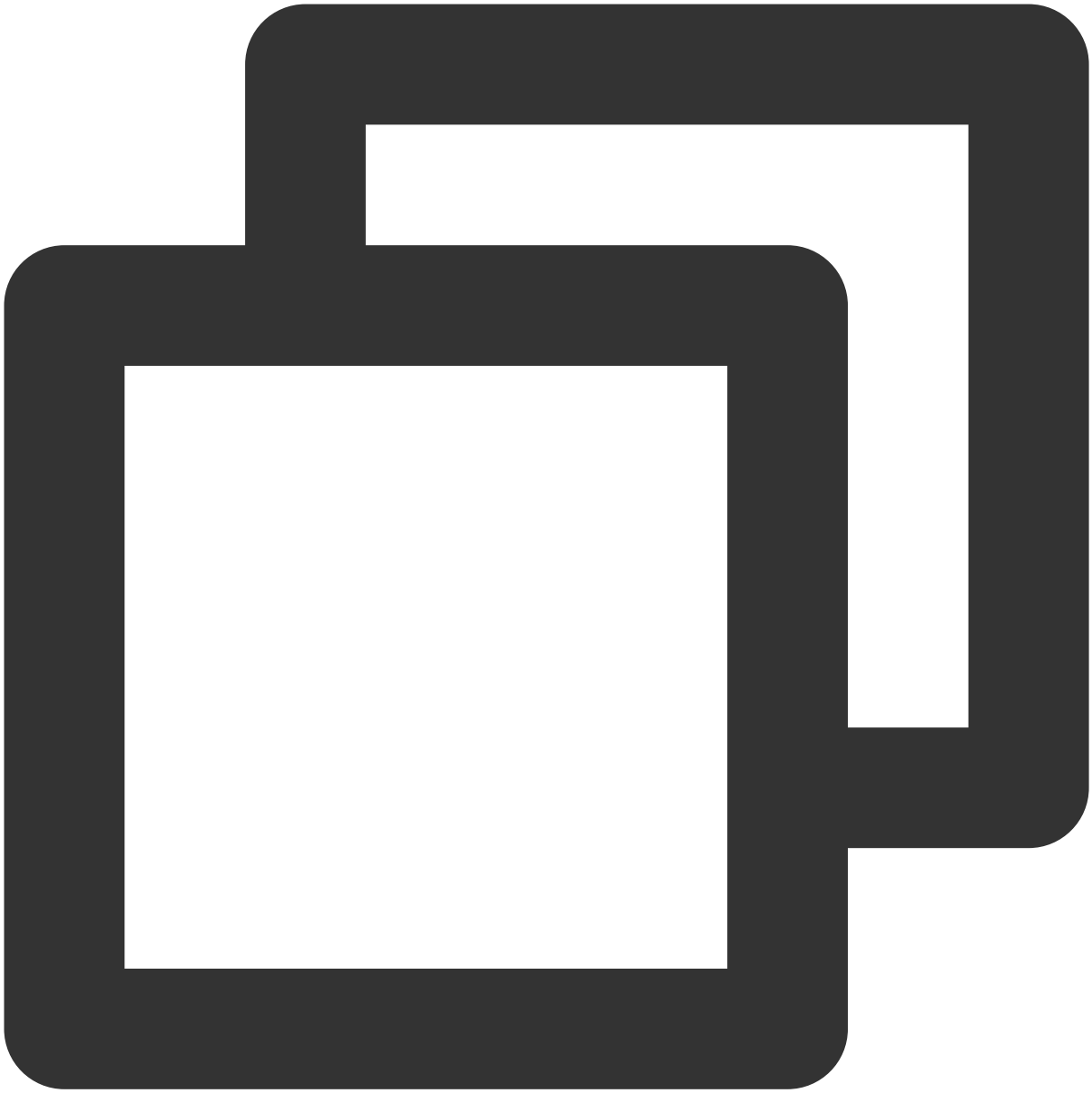
Note:

You need to enter the complete path of the topic name, i.e., `persistent://clusterid/namespace/Topic`, where the `clusterid/namespace/topic` part can be copied directly from the Topic Management page in the console.



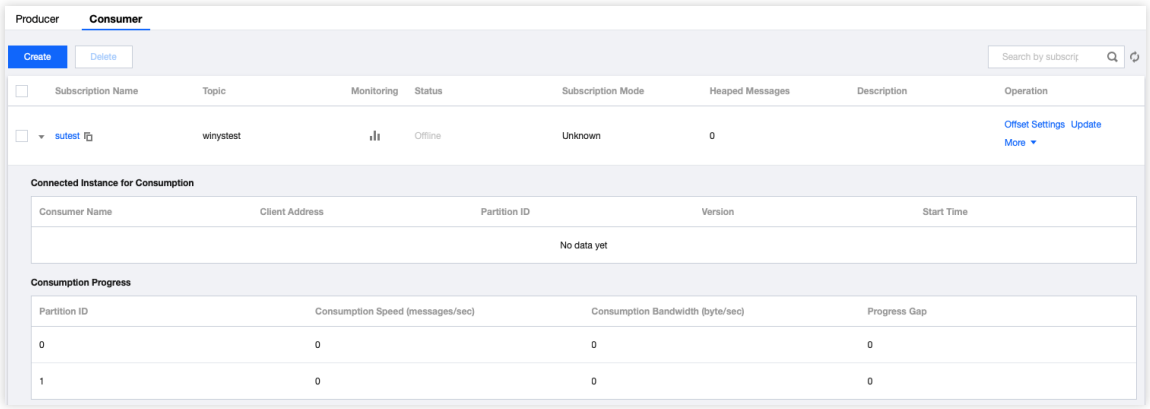
You need to enter the subscription name in the `subscriptionName` parameter, which can be viewed on the **Consumption Management** page.

6. Consume a message.



```
// Obtain the message
msg, err := consumer.Receive(context.Background())
if err != nil{
    log.Fatal(err)
}
// Simulate business processing
fmt.Printf("Received message msgId: %#v -- content: '%s'\\n",
           msg.ID(), string(msg.Payload()))

// If the consumption is successful, return `ack`; otherwise, return `nack` or `Re
consumer.Ack(msg)
```

7. Log in to the TDMQ for Apache Pulsar console, click **Topic** > **Topic Name** to enter the consumption management page, and click the triangle below a subscription name to view the production and consumption records.



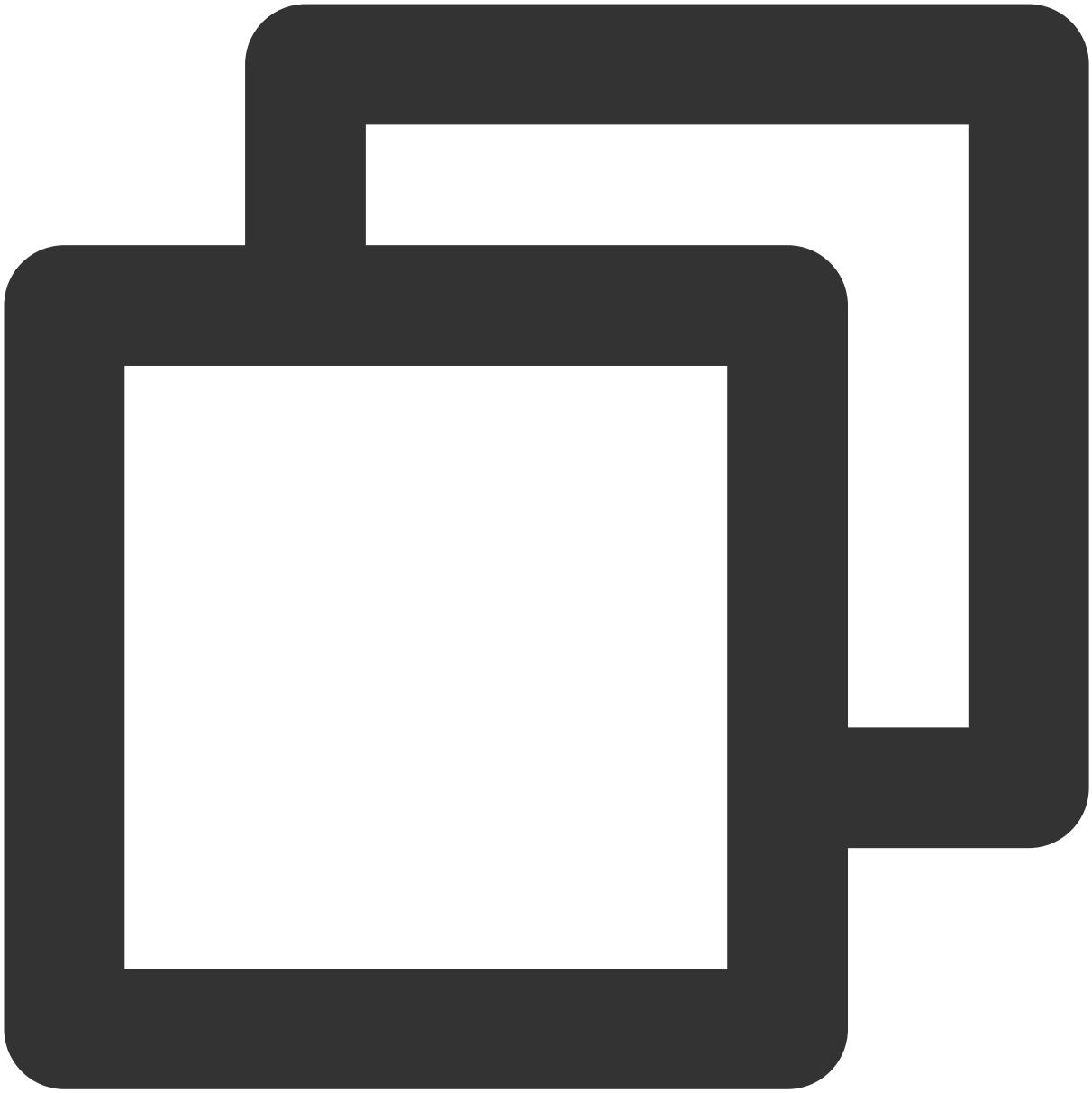
Note:

The above is a brief introduction to the way of publishing and subscribing to messages. For more operations, see Demo or Pulsar Go client.

Customizing Log File Output

Use Cases

As many users don't customize the logging library when using the Pulsar SDK for Go, logs are output to `os.Stderr` by default, as shown below:



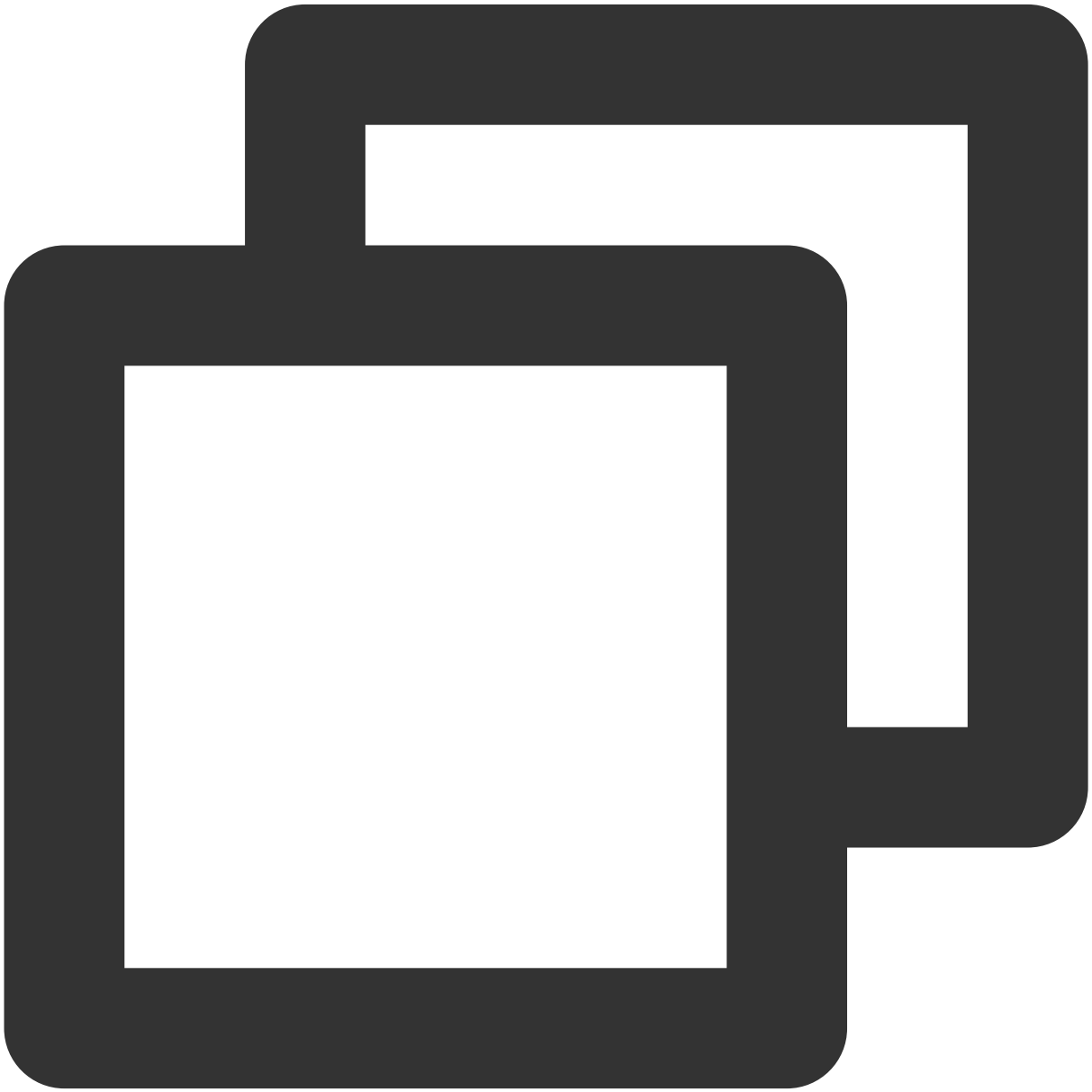
```
// It's recommended to make this a global instance called `log`.
func New() *Logger {
    return &Logger{
        Out:          os.Stderr, // Default output address
        Formatter:    new(TextFormatter),
        Hooks:        make(LevelHooks),
        Level:        InfoLevel,
        ExitFunc:     os.Exit,
        ReportCaller: false,
    }
}
```

Generally, log information is output to `os.Stderr`. If you don't specify a custom logging library, the SDK for Go logs and business logs will be mixed, making it difficult for troubleshooting.

Solution

With the `logger` API exposed on the client by the SDK for Go, you can customize the log output format and location and use logging libraries such as `logrus` and `zap`. Related parameters are as follows:

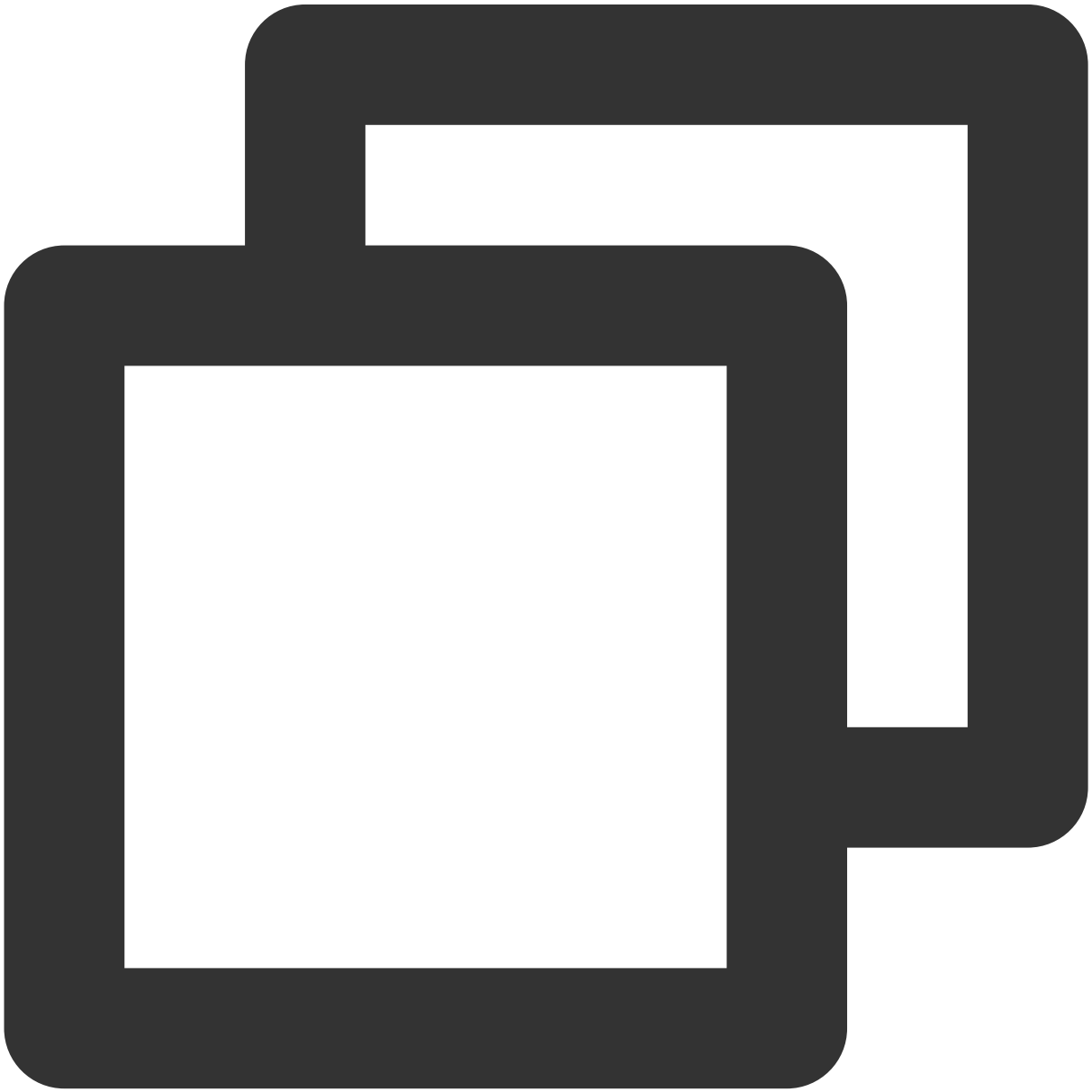
1. Implement the `log.Logger` API provided by the Pulsar SDK for Go by customizing `log lib`.



```
// ClientOptions is used to construct a Pulsar Client instance.
```

```
type ClientOptions struct {
    // Configure the logger used by the client.
    // By default, a wrapped logrus.StandardLogger will be used, namely,
    // log.NewLoggerWithLogrus(logrus.StandardLogger())
    // FIXME: use `logger` as internal field name instead of `log` as it's more idi
    Logger log.Logger
}
```

When using the SDK for Go, you can customize the `logger` API to customize `log lib` so that you can redirect logs to a specified location. Taking `logrus` as an example, the demo below shows you how to customize `log lib` to output the SDK for Go logs to a specified file.

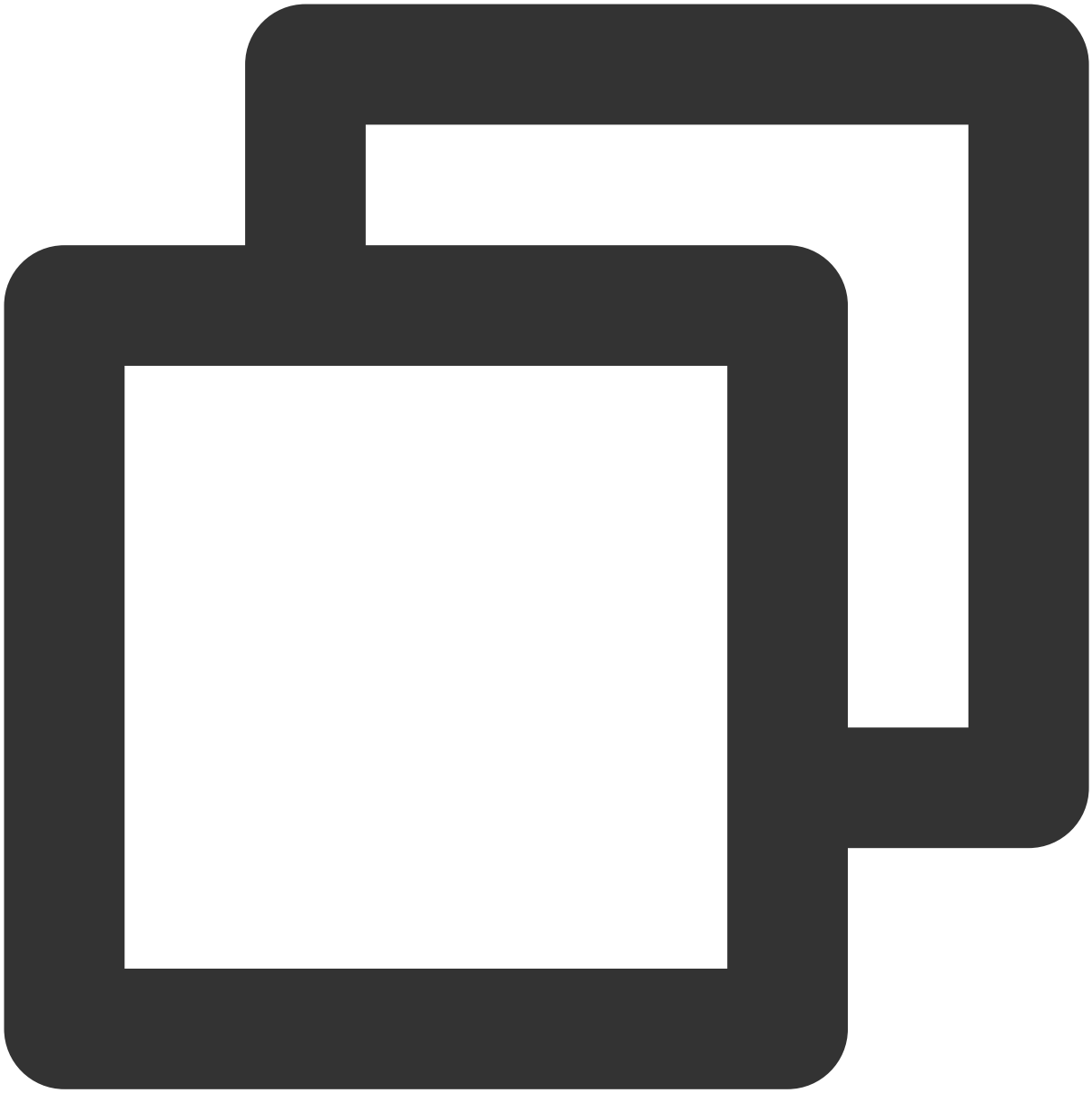


```
package main

import(
    "fmt"
    "io"
    "os"

    "github.com/apache/pulsar-client-go/pulsar/log"
    "github.com/sirupsen/logrus"

)

// logrusWrapper implements Logger interface
// based on underlying logrus.FieldLogger
type logrusWrapper struct {
    l logrus.FieldLogger
}

// NewLoggerWithLogrus creates a new logger which wraps
// the given logrus.Logger
func NewLoggerWithLogrus(logger *logrus.Logger, outputPath string) log.Logger {
    writer1 := os.Stdout
    writer2, err := os.OpenFile(outputPath, os.O_WRONLY|os.O_CREATE, 0755)
    if err != nil{
        logrus.Error("create file log.txt failed: %v", err)
    }
    logger.SetOutput(io.MultiWriter(writer1, writer2))
    return &logrusWrapper{
        l: logger,
    }
}

func (l *logrusWrapper) SubLogger(fs log.Fields) log.Logger {
    return &logrusWrapper{
        l: l.l.WithFields(logrus.Fields(fs)),
    }
}

func (l *logrusWrapper) WithFields(fs log.Fields) log.Entry {
    return logrusEntry{
        e: l.l.WithFields(logrus.Fields(fs)),
    }
}

func (l *logrusWrapper) WithField(name string, value interface{}) log.Entry {
    return logrusEntry{
```

```
        e: l.l.WithField(name, value),
    }
}

func (l *logrusWrapper) WithError(err error) log.Entry {
    return logrusEntry{
        e: l.l.WithError(err),
    }
}

func (l *logrusWrapper) Debug(args ...interface{}) {
    l.l.Debug(args...)
}

func (l *logrusWrapper) Info(args ...interface{}) {
    l.l.Info(args...)
}

func (l *logrusWrapper) Warn(args ...interface{}) {
    l.l.Warn(args...)
}

func (l *logrusWrapper) Error(args ...interface{}) {
    l.l.Error(args...)
}

func (l *logrusWrapper) Debugf(format string, args ...interface{}) {
    l.l.Debugf(format, args...)
}

func (l *logrusWrapper) Infof(format string, args ...interface{}) {
    l.l.Infof(format, args...)
}

func (l *logrusWrapper) Warnf(format string, args ...interface{}) {
    l.l.Warnf(format, args...)
}

func (l *logrusWrapper) Errorf(format string, args ...interface{}) {
    l.l.Errorf(format, args...)
}

type logrusEntry struct {
    e logrus.FieldLogger
}

func (l logrusEntry) WithFields(fs log.Fields) log.Entry {
```

```
    return logrusEntry{
        e: l.e.WithFields(logrus.Fields(fs)),
    }
}

func (l logrusEntry) WithField(name string, value interface{}) log.Entry {
    return logrusEntry{
        e: l.e.WithField(name, value),
    }
}

func (l logrusEntry) Debug(args ...interface{}) {
    l.e.Debug(args...)
}

func (l logrusEntry) Info(args ...interface{}) {
    l.e.Info(args...)
}

func (l logrusEntry) Warn(args ...interface{}) {
    l.e.Warn(args...)
}

func (l logrusEntry) Error(args ...interface{}) {
    l.e.Error(args...)
}

func (l logrusEntry) Debugf(format string, args ...interface{}) {
    l.e.Debugf(format, args...)
}

func (l logrusEntry) Infof(format string, args ...interface{}) {
    l.e.Infof(format, args...)
}

func (l logrusEntry) Warnf(format string, args ...interface{}) {
    l.e.Warnf(format, args...)
}

func (l logrusEntry) Errorf(format string, args ...interface{}) {
    l.e.Errorf(format, args...)
}
```

2. Specify a custom `log lib` when creating the client.



```
client, err := pulsar.NewClient(pulsar.ClientOptions{
    URL:    "pulsar://localhost:6650",
    Logger: NewLoggerWithLogrus(log.StandardLogger(), "test.log"),
})
```

The above demo shows you how to redirect the log file of the Pulsar SDK for Go to the `test.log` file in the current path. You can redirect the log file to a specified location as needed.

SDK for C++

Last updated：2024-06-28 11:33:56

Overview

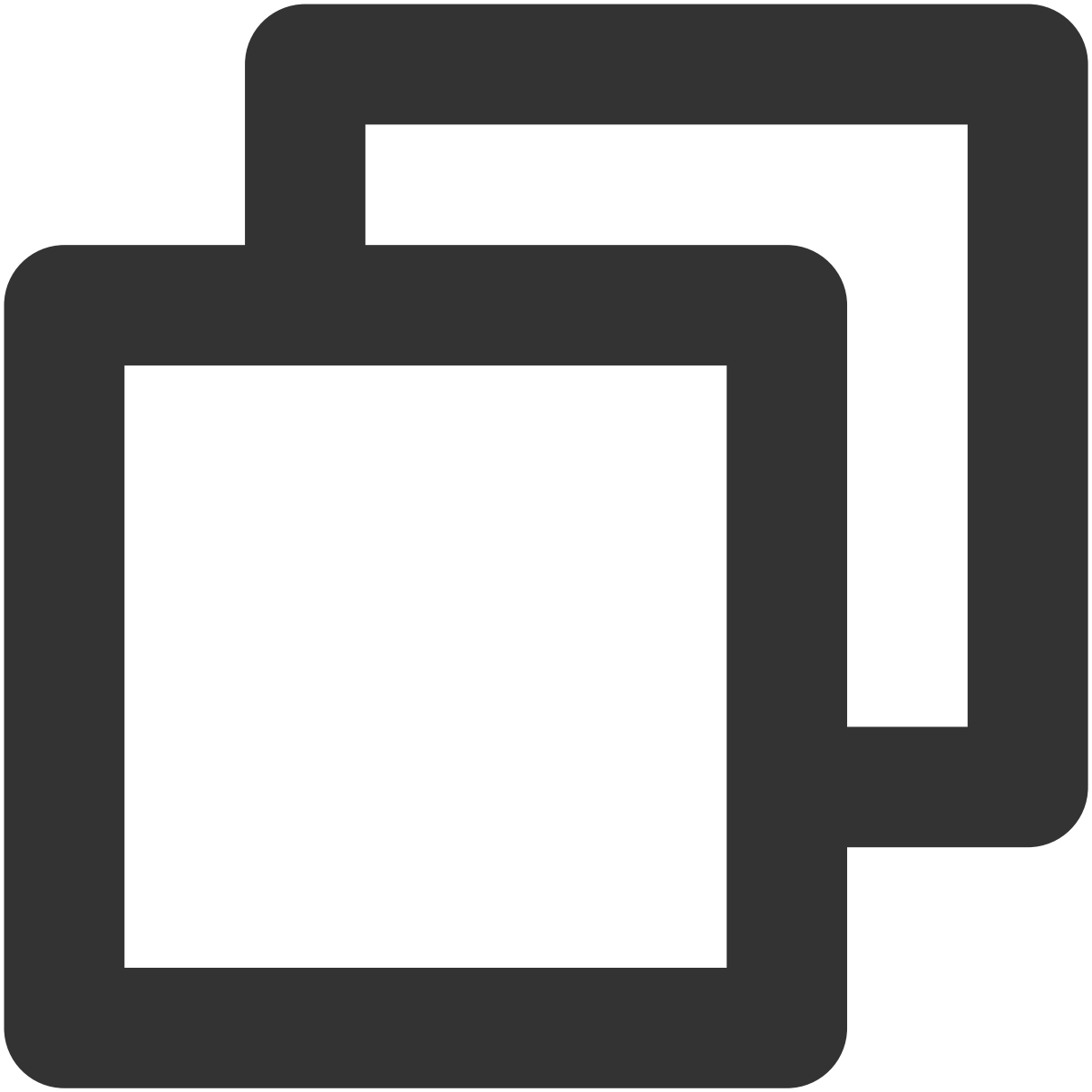
This document describes how to use open-source SDK to send and receive messages by using the SDK for C++ as an example and helps you better understand the message sending and receiving processes.

Prerequisites

You have created the required resources.
You have installed GCC
You have downloaded the demo

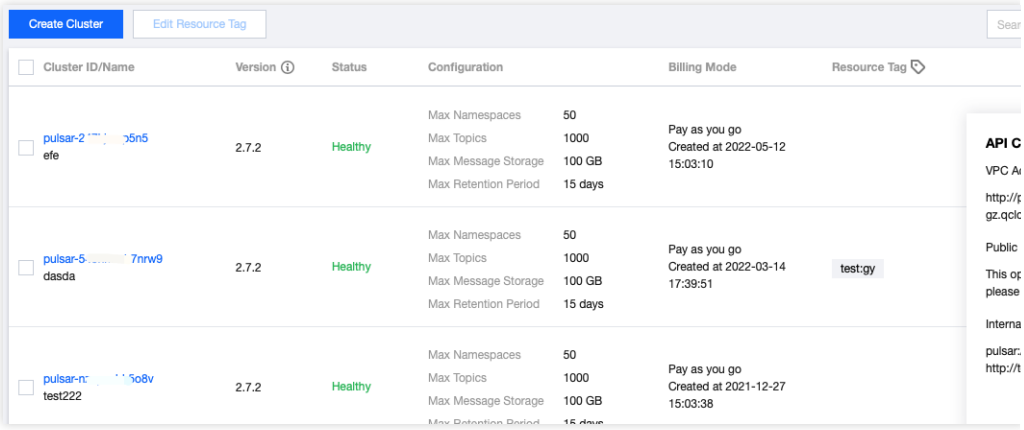
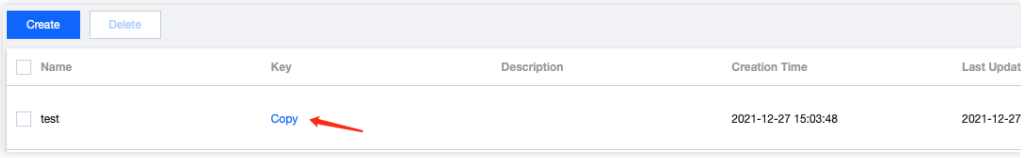
Directions

1. Prepare the environment.
1.1 Install the Pulsar C++ client in the client environment as instructed in Pulsar C++ client.
1.2 Introduce the files and dynamic libraries that are related to the Pulsar C++ client to the project.
2. Create a client.

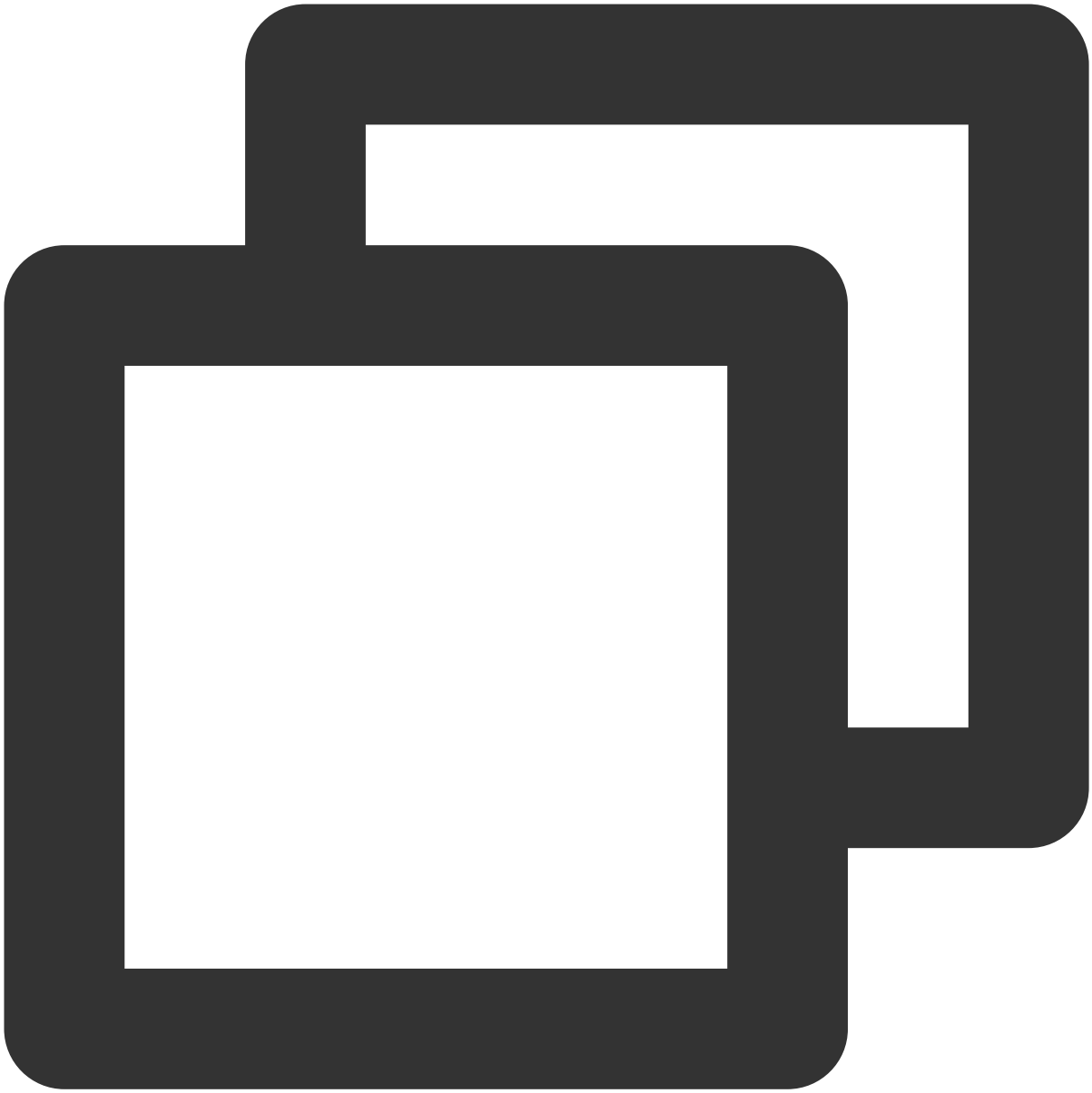


```
// Client configuration information
ClientConfiguration config;
// Set the role token
AuthenticationPtr auth = pulsar::AuthToken::createWithToken(AUTHENTICATION);
config.setAuth(auth);
// Create a client
Client client(SERVICE_URL, config);
```

Parameter	Description

SERVICE_URL	Cluster access address, which can be viewed and copied on the Cluster page in the consol 
AUTHENTICATION	Role token, which can be copied in the Token column on the Role Management page. 

3. Create a producer.



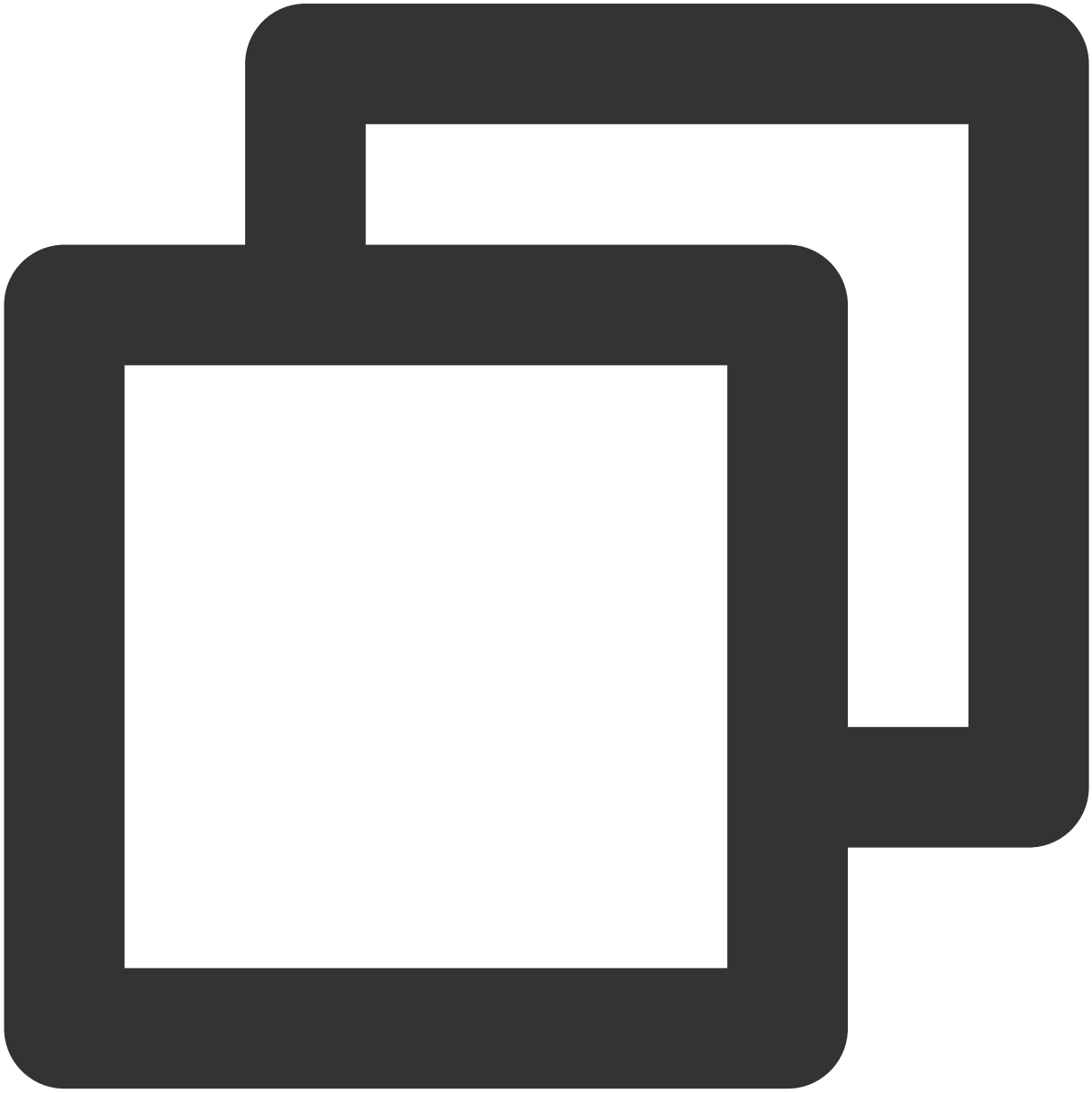
```
// Producer configurations
ProducerConfiguration producerConf;
producerConf.setBlockIfQueueFull(true);
producerConf.setSendTimeout(5000);
// Producer
Producer producer;
// Create a producer
Result result = client.createProducer(
    // Complete path of the topic in the format of `persistent://cluster (tenant) I
    "persistent://pulsar-xxx/sdk_cpp/topic1",
    producerConf,
```

```
    producer);
if (result != ResultOk) {
    std::cout << "Error creating producer: " << result << std::endl;
    return -1;
}
```

Note:

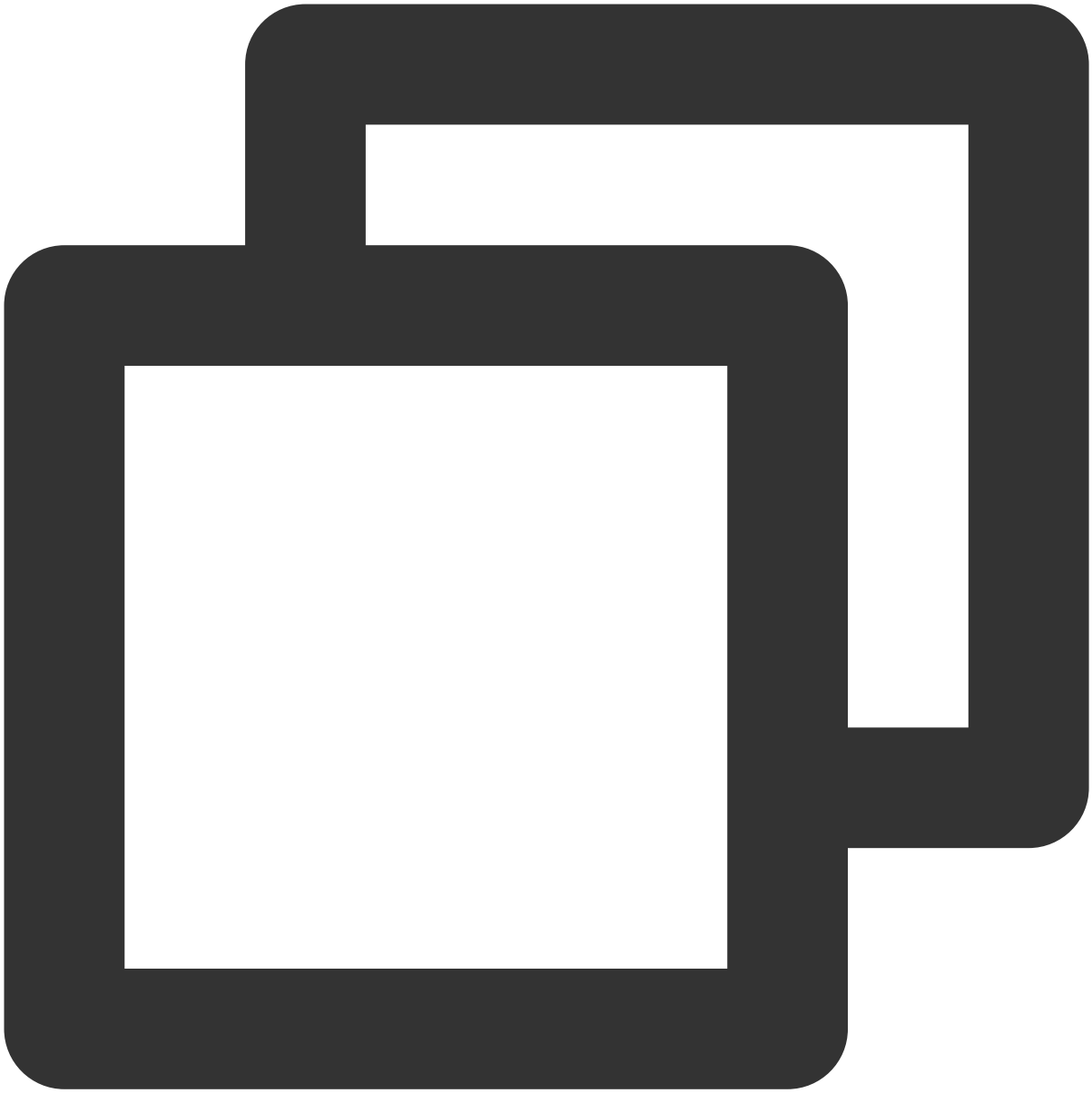
You need to enter the complete path of the topic name, i.e., `persistent://clusterid/namespace/Topic`, where the `clusterid/namespace/topic` part can be copied directly from the **Topic** page in the console.

4. Send the message.



```
// Message content
std::string content = "hello cpp client, this is a msg";
// Create a message object
Message msg = MessageBuilder().setContent(content)
    .setPartitionKey("mykey")  // Business key
    .setProperty("x", "1")   // Set message parameters
    .build();
// Send the message
Result result = producer.send(msg);
if (result != ResultOk) {
    // The message failed to be sent
    std::cout << "The message " << content << " could not be sent, received code
} else {
    // The message was successfully sent
    std::cout << "The message " << content << " sent successfully" << std::endl;
}
```

5. Create a consumer.

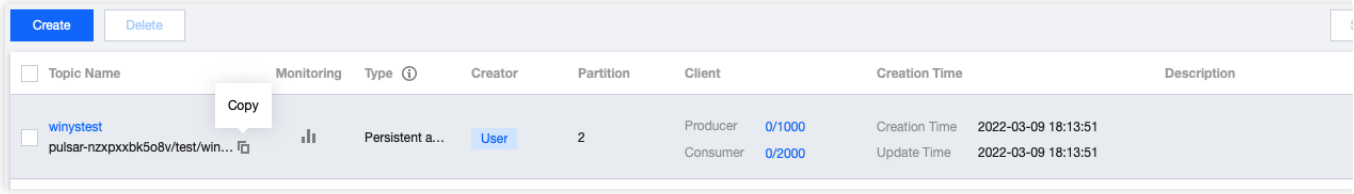


```
// Consumer configuration information
ConsumerConfiguration consumerConfiguration;
consumerConfiguration.setSubscriptionInitialPosition(pulsar::InitialPositionEarl
// Consumer
Consumer consumer;
// Subscribe to a topic
Result result = client.subscribe(
    // Complete path of the topic in the format of `persistent://cluster (tenant
    "persistent://pulsar-xxx/sdk_cpp/topic1",
    // Subscription name
    "sub_topic1",
```

```
        consumerConfiguration,
        consumer);

    if (result != ResultOk) {
        std::cout << "Failed to subscribe: " << result << std::endl;
        return -1;
    }
```

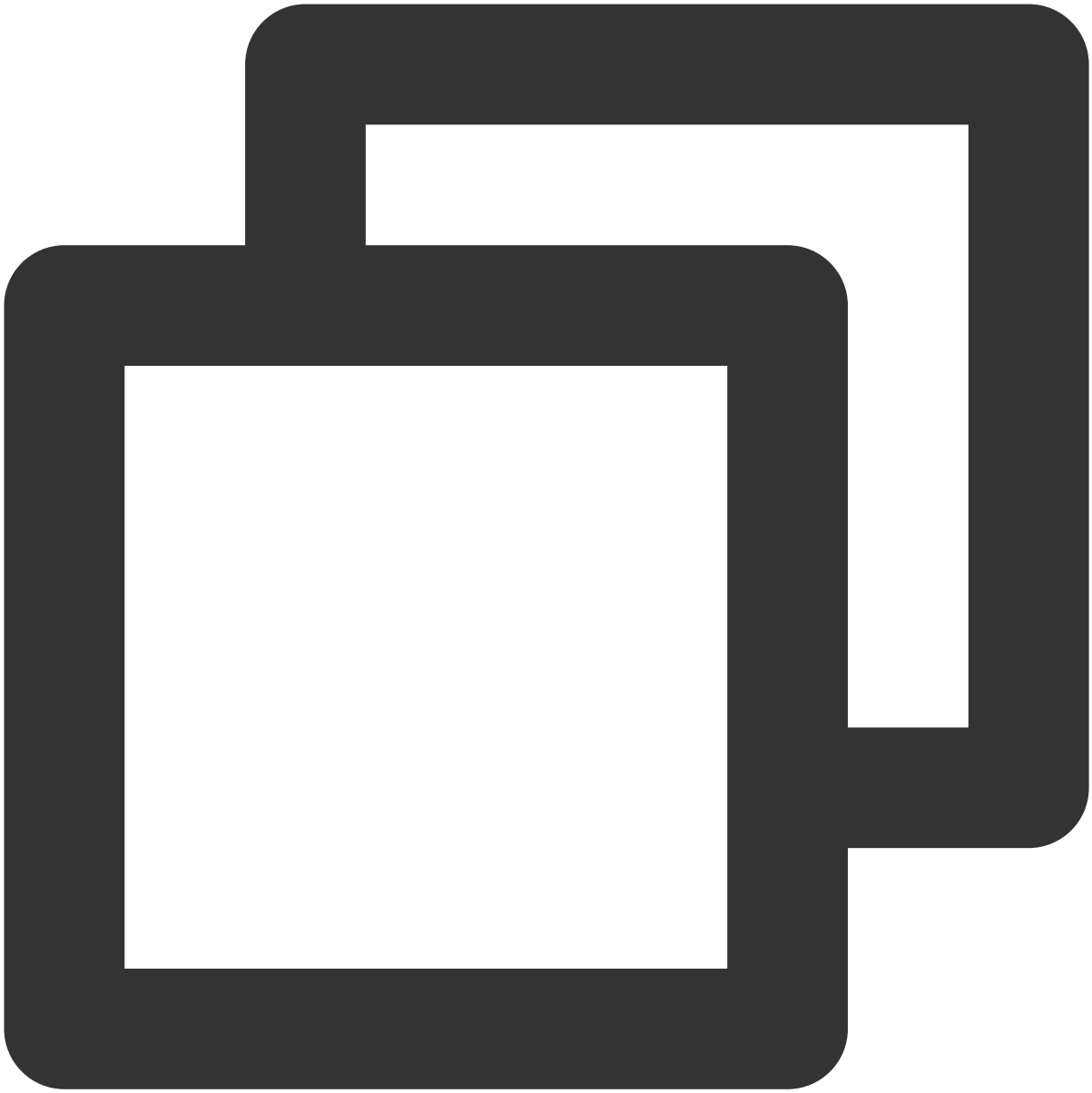
Note:

You need to enter the complete path of the topic name, i.e., `persistent://clusterid/namespace/Topic`, where the `clusterid/namespace/topic` part can be copied directly from the **Topic** page in the console.



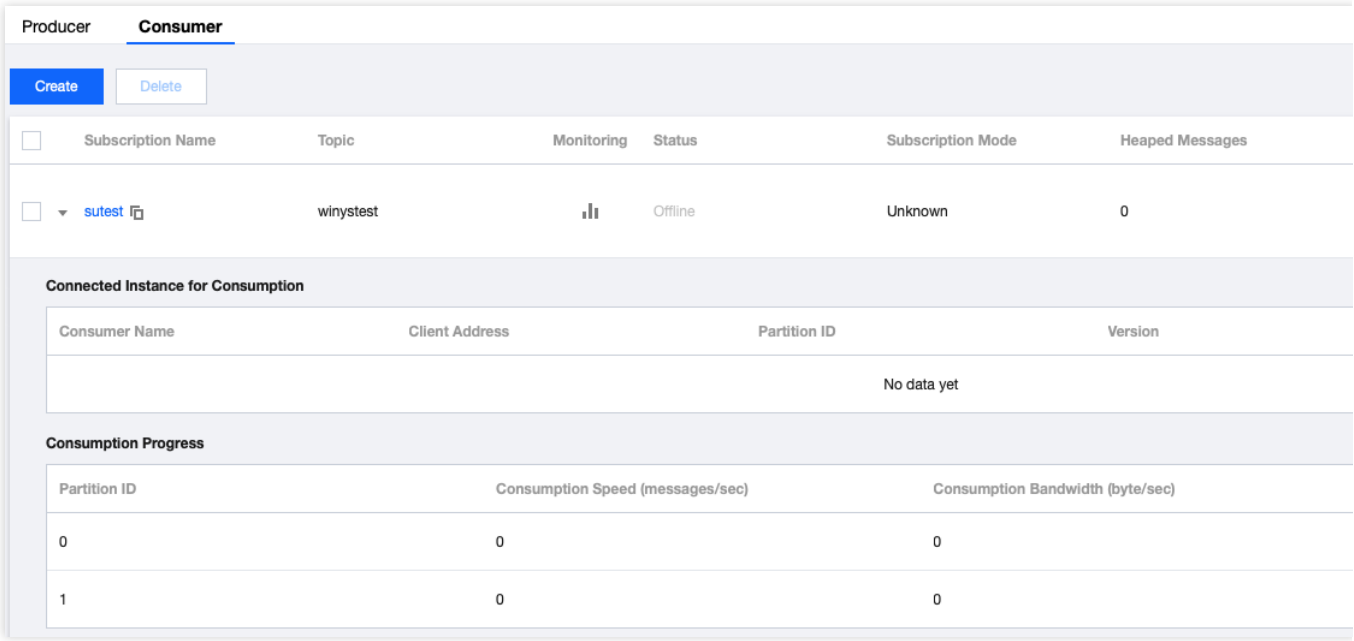
You need to enter the subscription name in the `subscriptionName` parameter, which can be viewed on the **Consumption Management** page.

6. Consume the message.



```
Message msg;
// Obtain the message
consumer.receive(msg);
// Simulate the business processing logic
std::cout << "Received: " << msg << "  with payload '" << msg.getDataAsString()
// Return `ack` as the acknowledgement
consumer.acknowledge(msg);
// Return `nack` if the consumption fails, and the message will be delivered aga
// consumer.negativeAcknowledge(msg);
```

7. Log in to the TDMQ for Apache Pulsar console, click **Topic** > **Topic Name** to enter the **Consumption Management** page, and click the triangle below a subscription name to view the production and consumption records.



Note:

The above is a brief introduction to the way of publishing and subscribing to messages. For more operations, see Demo or Pulsar C++ client.

SDK for Python

Last updated：2024-06-28 11:33:56

Overview

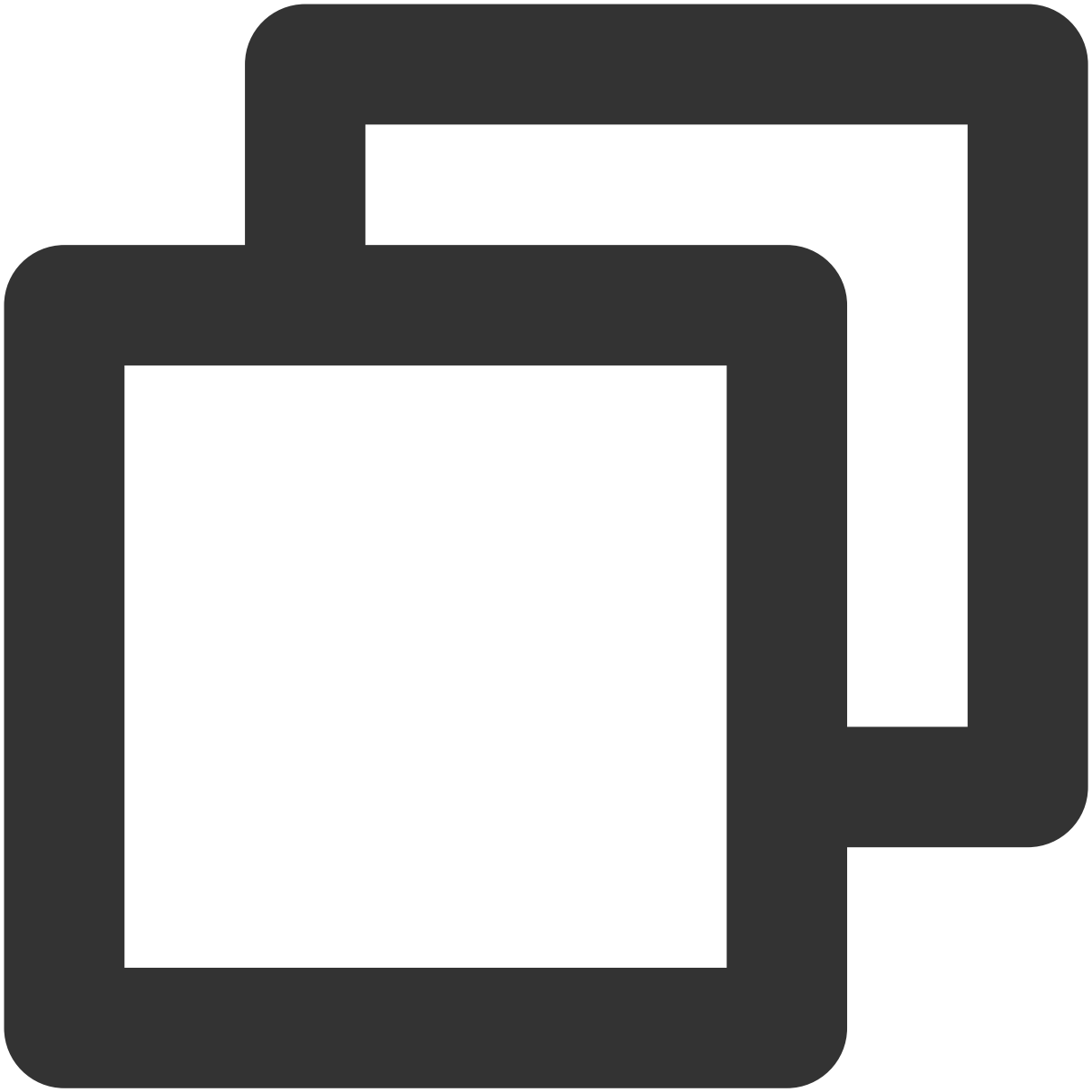
This document describes how to use open-source SDK to send and receive messages by using the SDK for Python as an example and helps you better understand the message sending and receiving processes.

Prerequisites

You have created or prepared the required resources as instructed in Resource Creation and Preparation.
You have installed Python. For the download address, click here.
You have installed pip. For the download address, click here.
You have downloaded the demo. For the download address, click here.

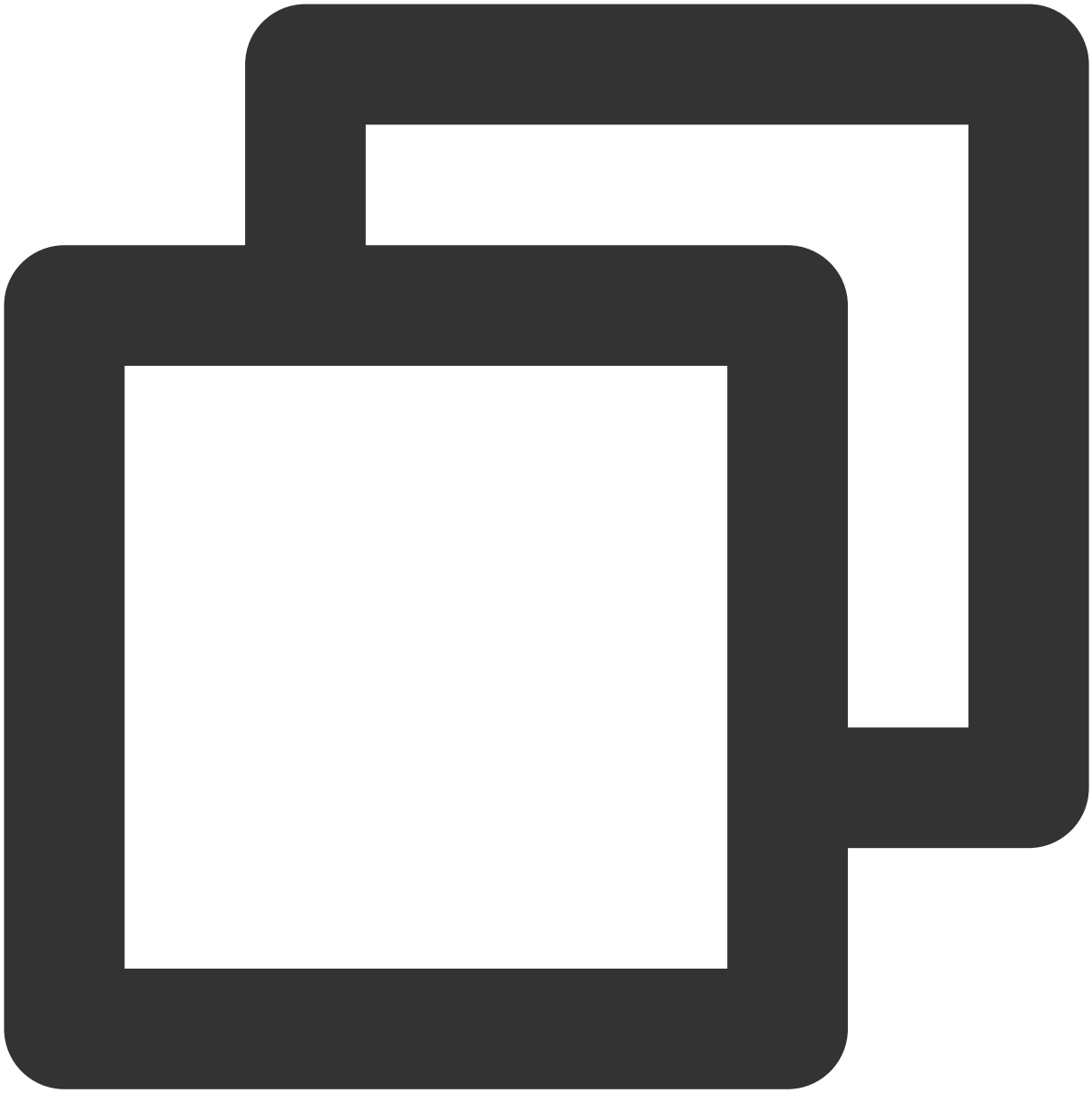
Directions

1. Prepare the environment.
Install the `pulsar-client` library in the client environment as instructed in Pulsar Python client.



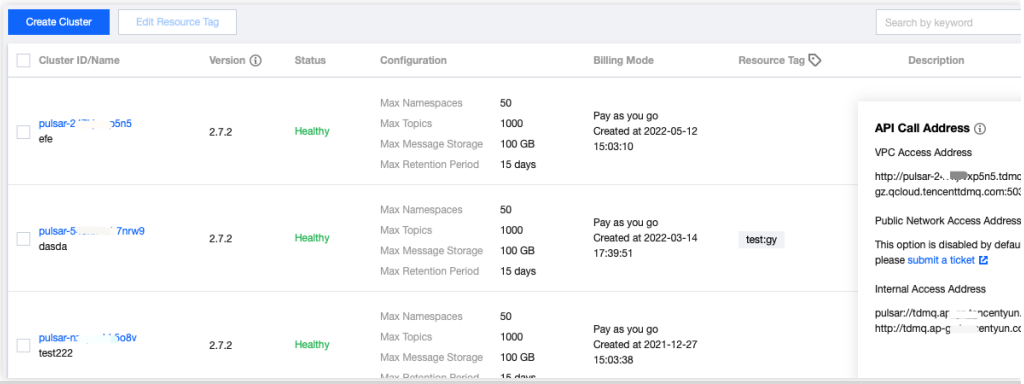
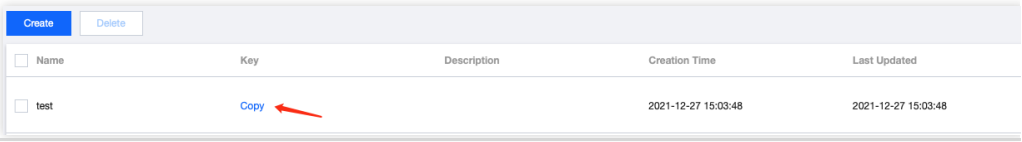
```
pip install 'pulsar-client==3.1.0'
```

2. Create a client.

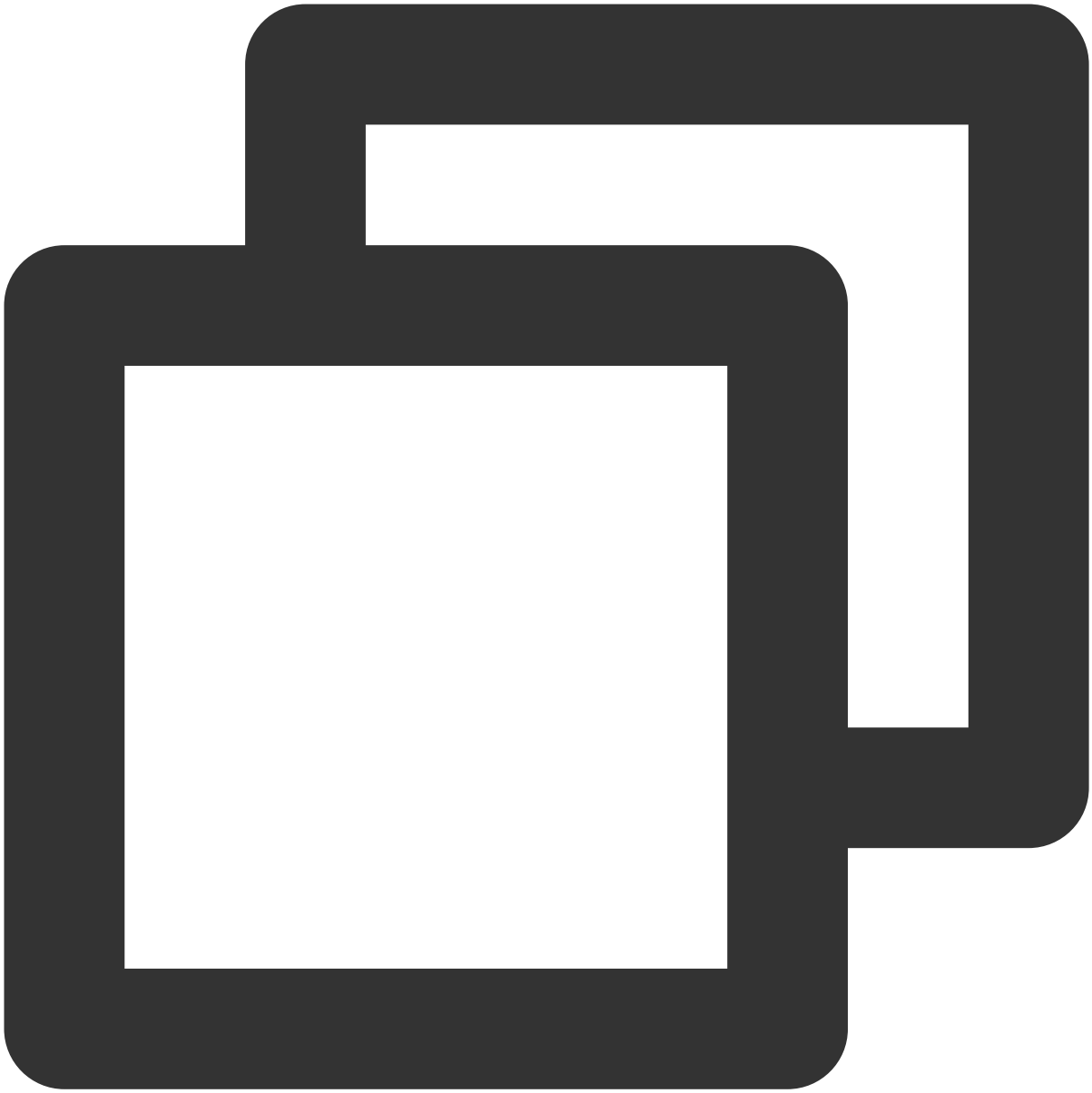


```
# Create a client
client = pulsar.Client(
    authentication=pulsar.AuthenticationToken(
        # Authorized role token
        AUTHENTICATION),
    # Service access address
    service_url=SERVICE_URL)
```

Parameter	Description

SERVICE_URL	Cluster access address, which can be viewed and copied on the Cluster page in the consol 
AUTHENTICATION	Role token, which can be copied in the **Token** column on the Role Management page. 

3. Create a producer.

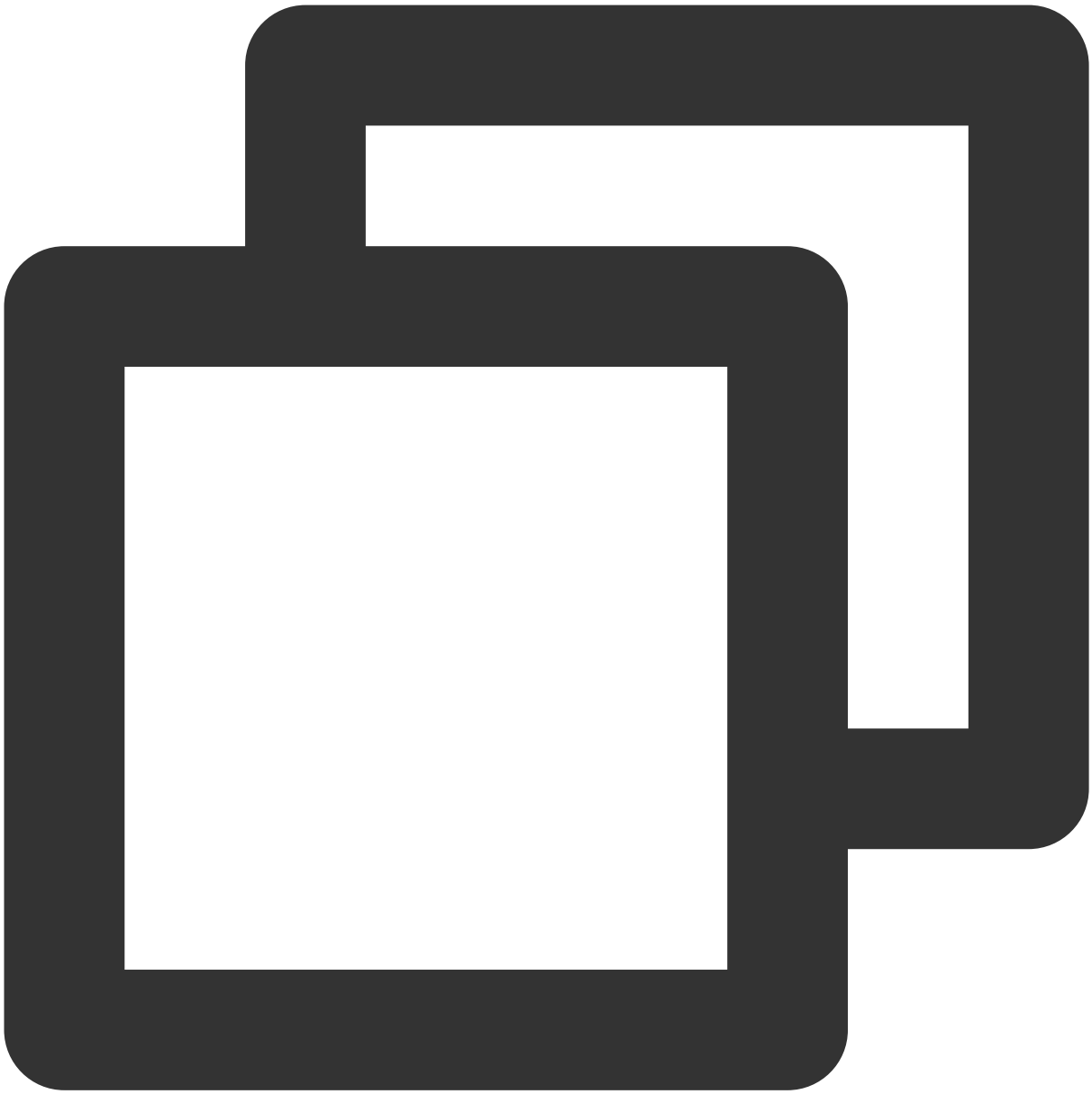


```
# Create a producer
producer = client.create_producer(
    # Complete path of the topic in the format of `persistent://cluster (tenant) I
    topic='pulsar-xxx/sdk_python/topic1'
)
```

Note:

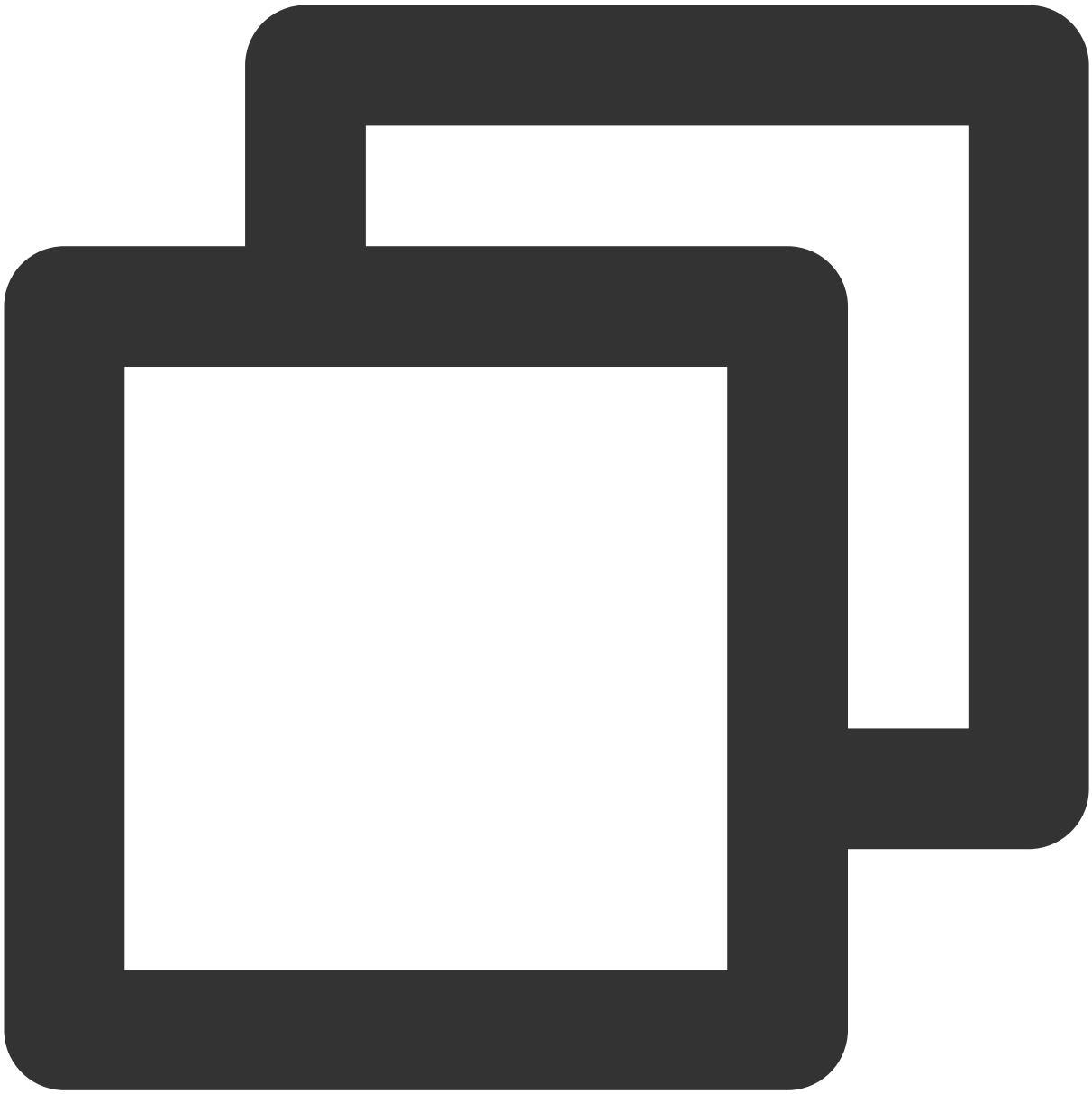
You need to enter the complete path of the topic name, that is, `persistent://clusterid/namespace/Topic` , where the `clusterid/namespace/topic` part can be copied directly from the Topic page in the console.

4. Send the message.



```
# Send the message
producer.send(
    # Message content
    'Hello python client, this is a msg.'.encode('utf-8'),
    # Message parameter
    properties={'k': 'v'},
    # Business key
    partition_key='yourKey'
)
```

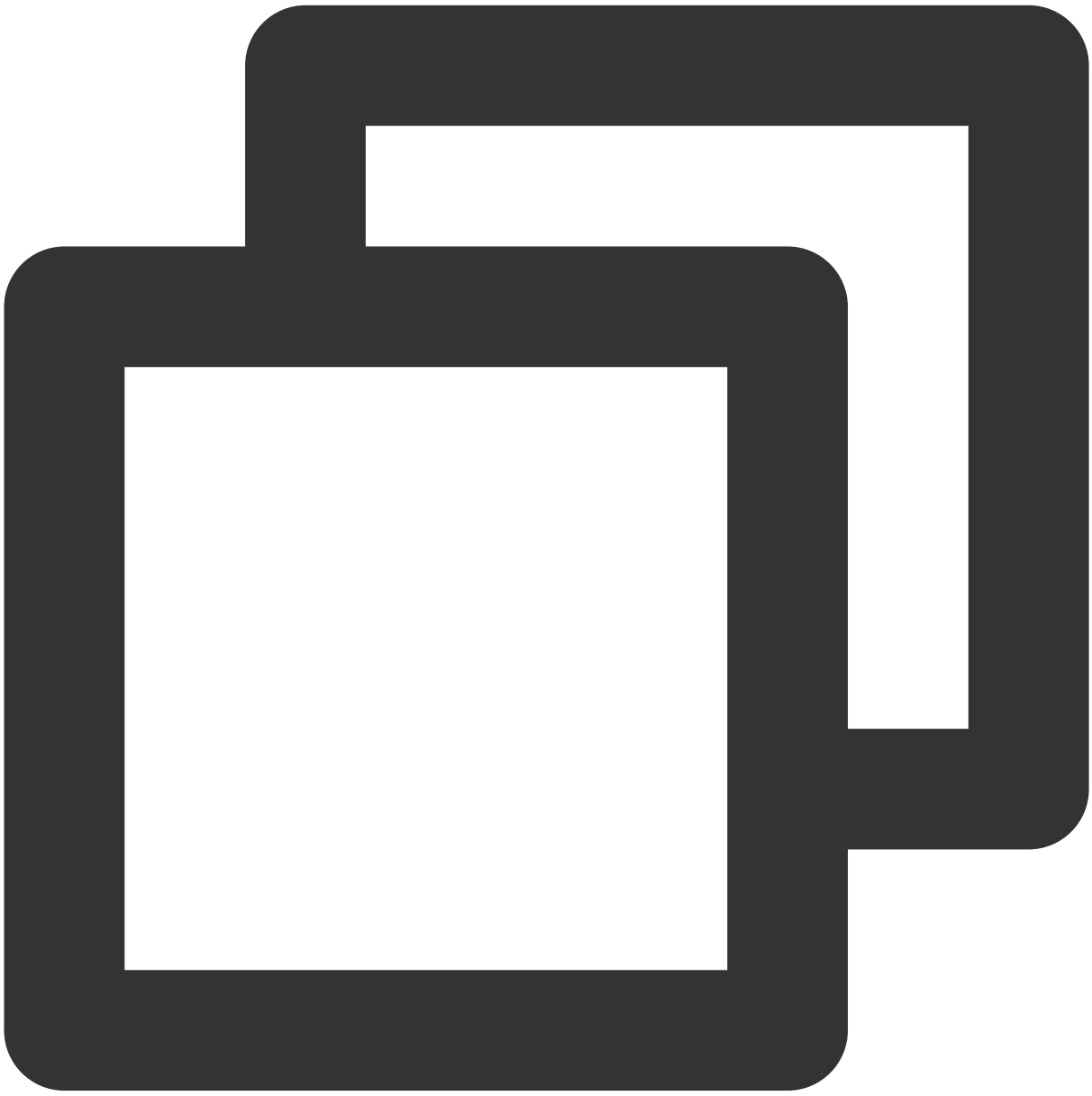
The message can also be sent in async mode.



```
# Send the callback in async mode
def send_callback(send_result, msg_id):
    print('Message published: result:{}  msg_id:{}'.format(send_result, msg_id))

# Send the message
producer.send_async(
    # Message content
    'Hello python client, this is a async msg.'.encode('utf-8'),
    # Async callback
```

```
    callback=send_callback,
    # Message configuration
    properties={'k': 'v'},
    # Business key
    partition_key='yourKey'
)
```

5. Create a consumer.

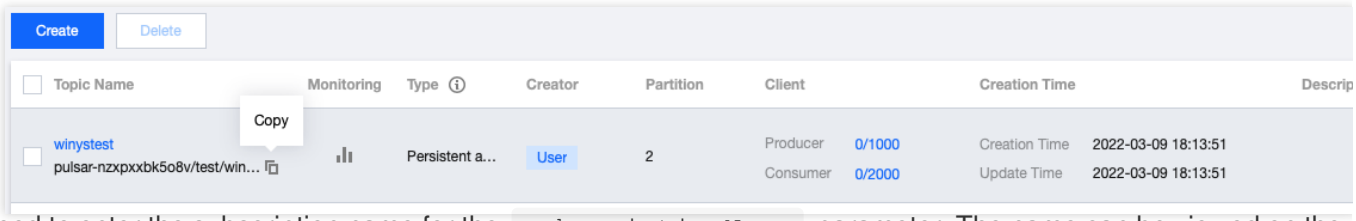


```
# Subscribe to the message
consumer = client.subscribe(
    # Complete path of the topic in the format of `persistent://cluster (tenant) I
```

```
        topic='pulsar-xxx/sdk_python/topic1',
        # Subscription name
        subscription_name='sub_topic1'
    )
```

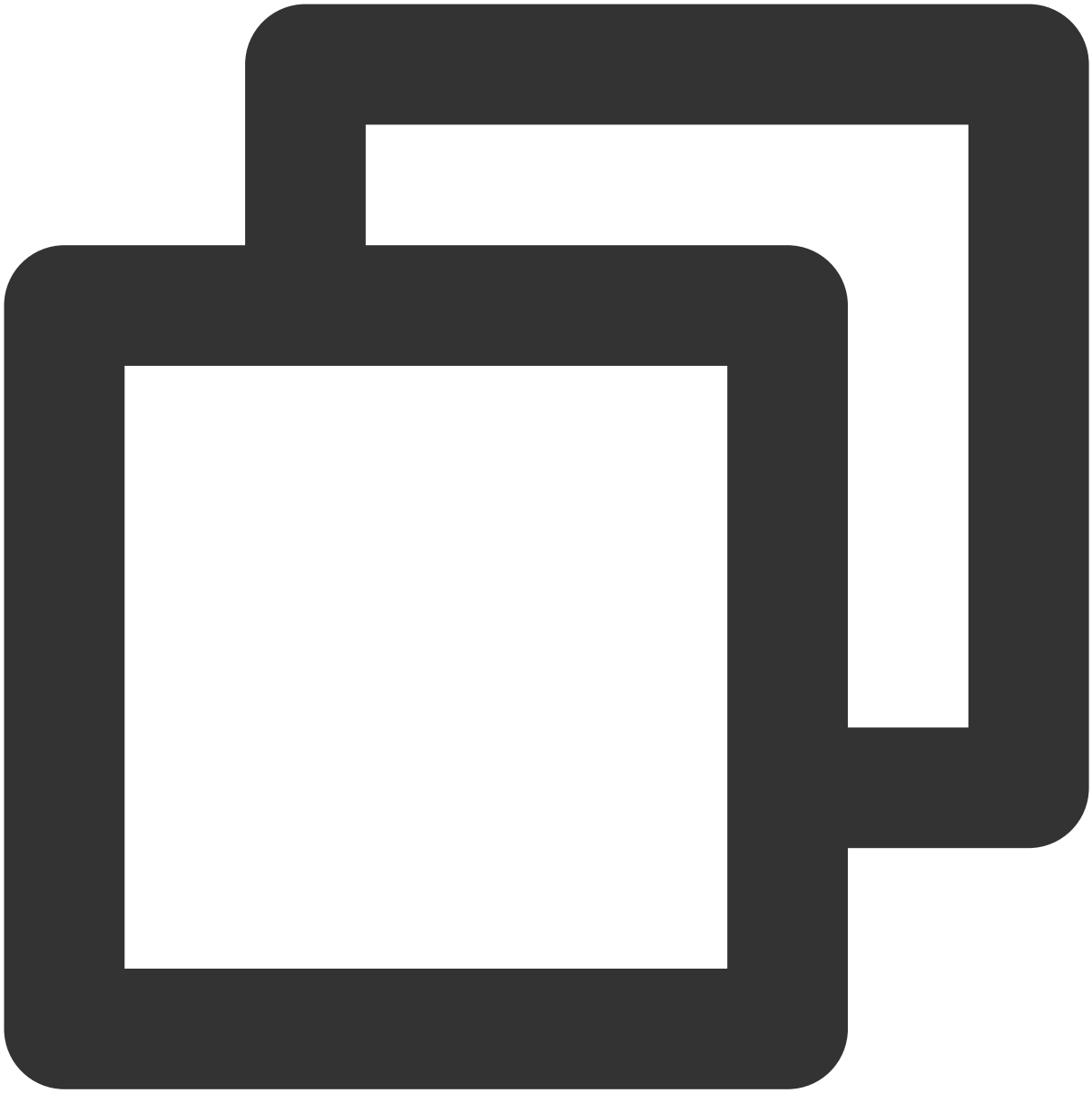
Note:

You need to enter the complete path of the topic name, that is, `persistent://clusterid/namespace/Topic`, where the `clusterid/namespace/topic` part can be copied directly from the Topic page in the console.



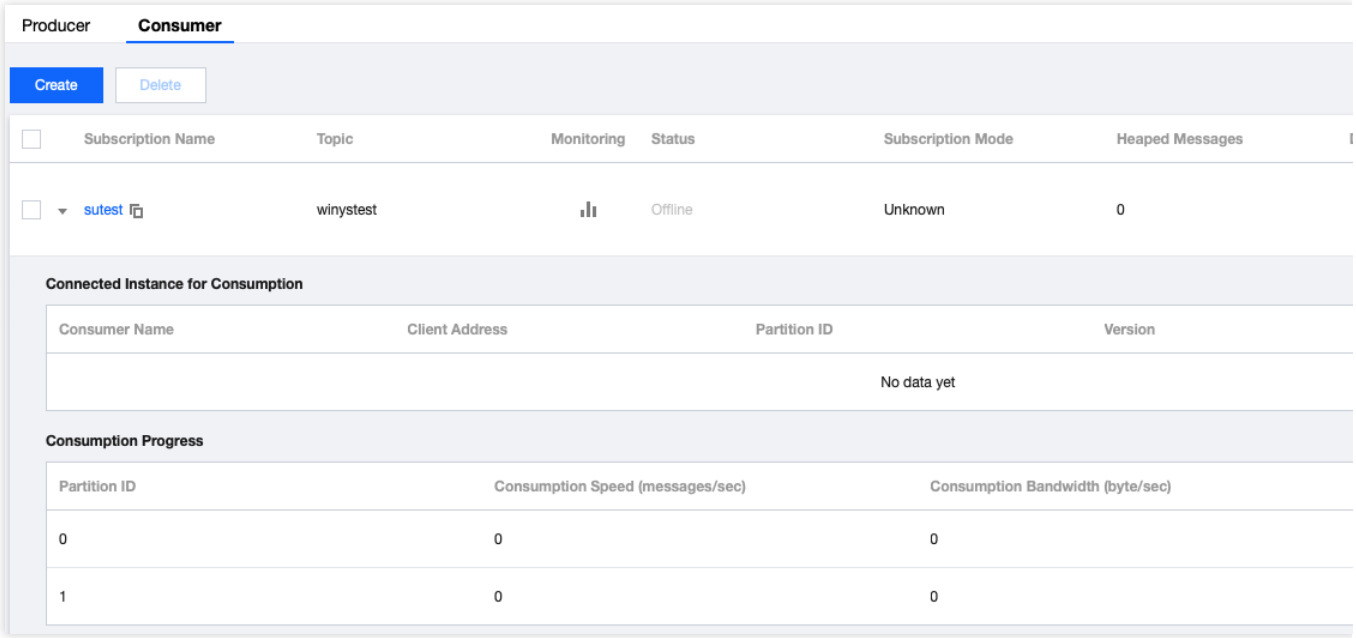
You need to enter the subscription name for the `subscriptionName` parameter. The name can be viewed on the **Consumption Management** page.

6. Consume the message.



```
# Obtain the message
msg = consumer.receive()
try:
    # Simulate the business processing logic
    print("Received message '{}' id='{}'".format(msg.data(), msg.message_id()))
    # Return `ack` as the acknowledgement if the consumption is successful
    consumer.acknowledge(msg)
except:
    # If the consumption fails, the message will be delivered again.
    consumer.negative_acknowledge(msg)
```

7. Log in to the TDMQ for Apache Pulsar console, click **Topic** > **Topic Name** to enter the consumption management page, and click the triangle below a subscription name to view the production and consumption records.



Note

Above is a brief introduction to message publishing and subscription. For more information, see Demo or Pulsar Python client.

SDK for Node.js

Last updated：2024-08-08 11:10:24

Overview

TDMQ for Apache Pulsar 2.7.1 and above clusters already support Apache Pulsar SDK for Node.js. This document describes how to access the SDK.

Prerequisites

Get the access address

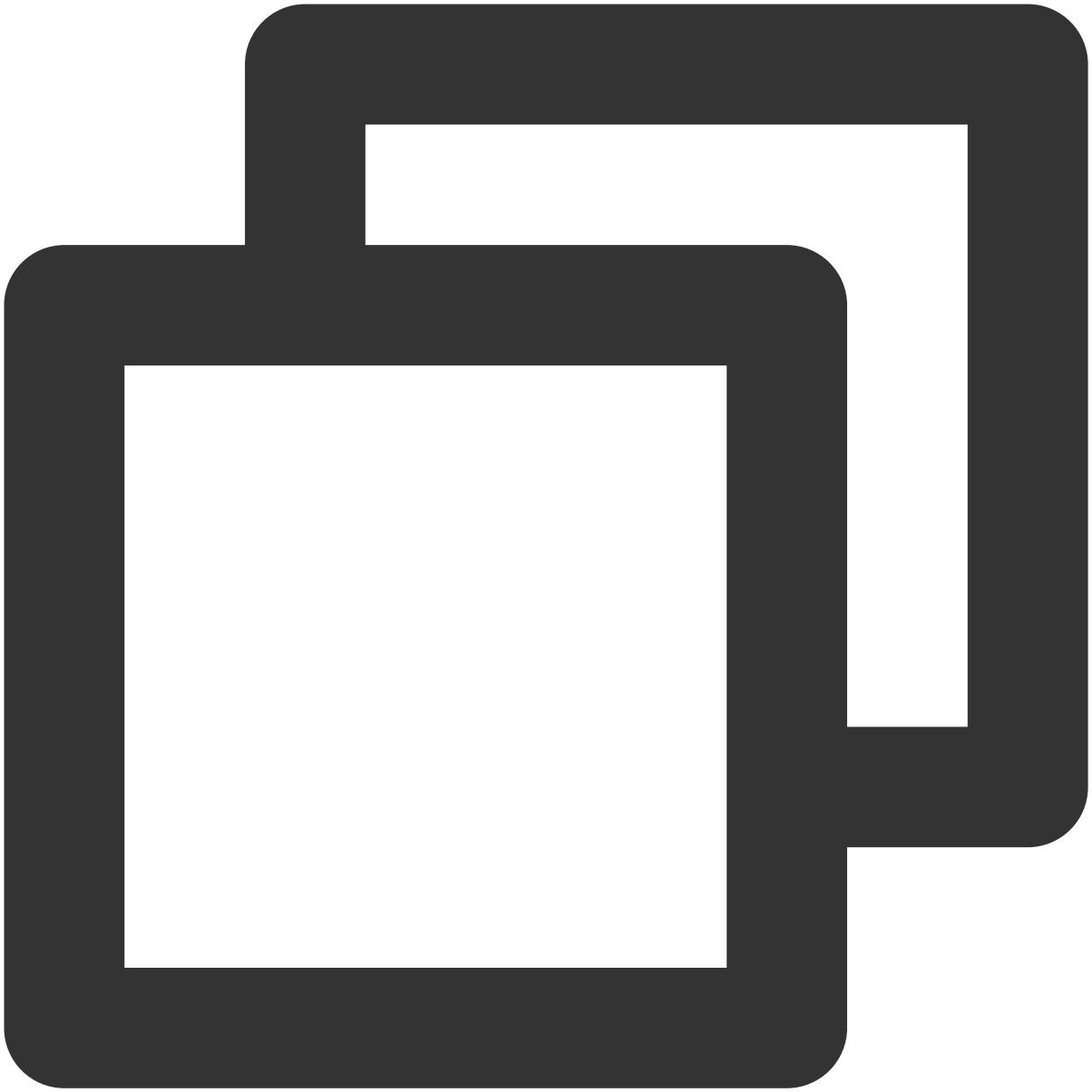
Copy the access address on the **Cluster Management** page in the TDMQ for Apache Pulsar console.

Get the token

Configure the role and permission as instructed in Role and Authentication and get the token of the role.

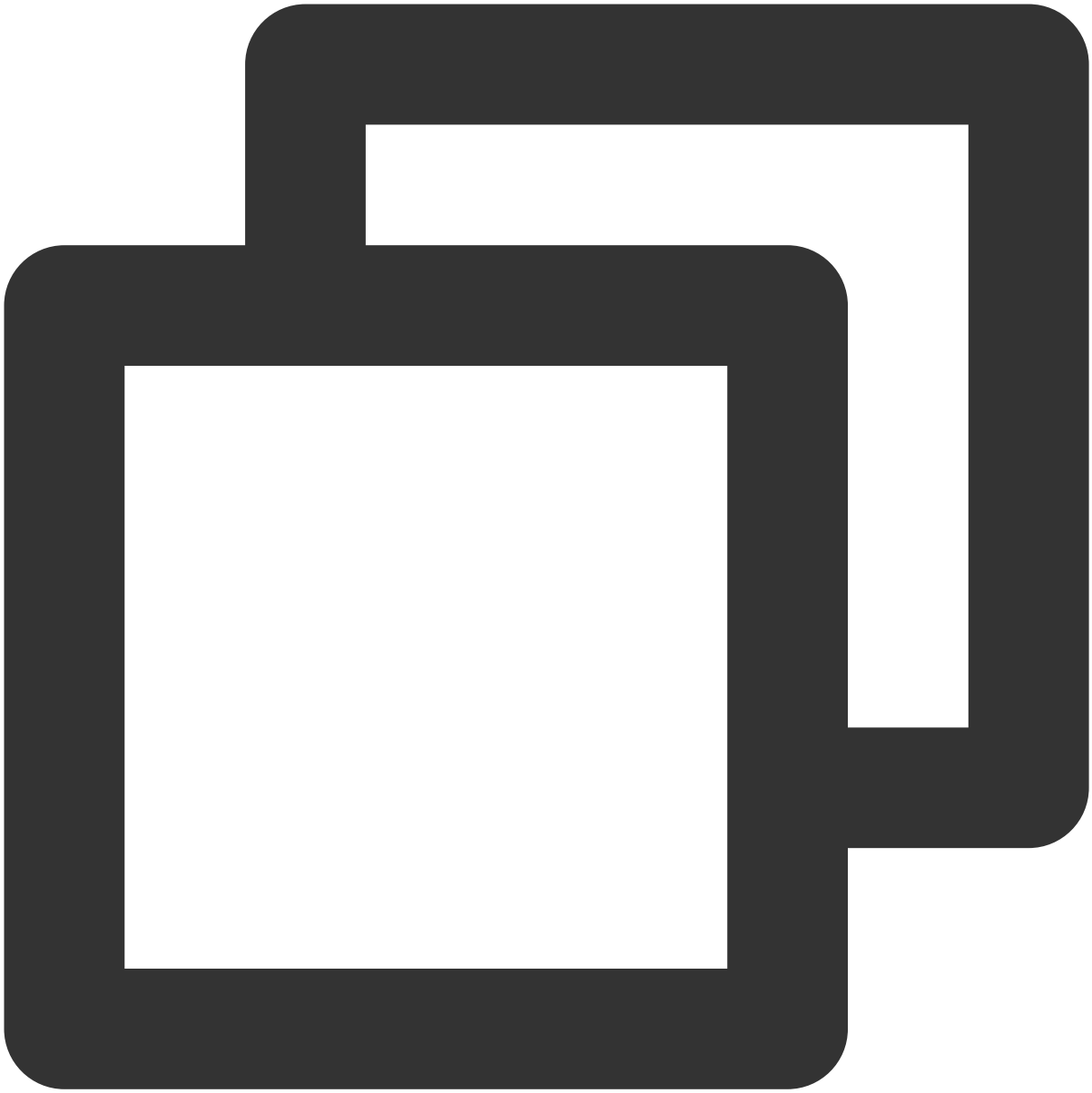
Directions

1. Install the Node.js client in your client environment as instructed in Pulsar Node.js client.



```
$ npm install pulsar-client
```

2. In the code for creating the Node.js client, configure the prepared access address and token.



```
const Pulsar = require('pulsar-client');

(async () => {
  const client = new Pulsar.Client({
    serviceUrl: 'http://*', // Replace with the access address (copied from the
    authentication:    Pulsar.NewAuthenticationToken("eyJh**"), // Replace with
  });

  await client.close();
})();
```

For how to use various features of the Apache Pulsar SDK for Node.js, see Pulsar Node.js client.