Client Libraries Java

Pulsar Java client

You can use a Pulsar Java client to create the Java producer, consumer, reader and TableView of messages and to perform administrative tasks. The current Java client version is **2.11.0**.

All the methods in producer, consumer, reader and TableView of a Java client are thread-safe.

Javadoc for the Pulsar client is divided into two domains by package as follows.

Package	Description	Maven Artifact
`org.apache.pulsar.client.api`	The producer and consumer API	org.apache.pulsar:pulsar-client:2.11.0
`org.apache.pulsar.client.admin`	The Java admin API	org.apache.pulsar:pulsar-client-admin:2.11.0
`org.apache.pulsar.client.all`	Include both `pulsar-client` and `pulsar-client-admin` Both `pulsar-client` and `pulsar-client-admin` are shaded packages and they shade dependencies independently. Consequently, the applications using both `pulsar-client` and `pulsar-client-admin` have redundant shaded classes. It would be troublesome if you introduce new dependencies but forget to update shading rules. In this case, you can use `pulsar-client-all`, which shades dependencies only one time and reduces the size of dependencies.	org.apache.pulsar:pulsar-client-all:2.11.0

This document focuses only on the client API for producing and consuming messages on Pulsar topics. For how to use the Java admin client, see Pulsar admin interface.

Installation

The latest version of the Pulsar Java client library is available via Maven Central. To use the latest version, add the `pulsar-client` library to your build configuration.

> **TIP**
>
> - `pulsar-client` and `pulsar-client-admin` shade dependencies via maven-shade-plugin to avoid conflicts of the underlying dependency packages (such as Netty). If you do not want to manage dependency conflicts manually, you can use them.
> - `pulsar-client-original` and `pulsar-client-admin-original` **does not** shade dependencies. If you want to manage dependencies manually, you can use them.

Maven

If you use Maven, add the following information to the `pom.xml` file.

```xml
<!-- in your <properties> block -->
<pulsar.version>2.11.0</pulsar.version>

<!-- in your <dependencies> block -->
<dependency>
  <groupId>org.apache.pulsar</groupId>
  <artifactId>pulsar-client</artifactId>
  <version>${pulsar.version}</version>
</dependency>
```

Gradle

If you use Gradle, add the following information to the `build.gradle` file.

```
def pulsarVersion = '2.11.0'

dependencies {
    compile group: 'org.apache.pulsar', name: 'pulsar-client', version: pulsarVersion
}
```

Connection URLs

To connect to Pulsar using client libraries, you need to specify a Pulsar protocol URL.

You can assign Pulsar protocol URLs to specific clusters and use the `pulsar` scheme. The following is an example of `localhost` with the default port `6650`:

```
pulsar://localhost:6650
```

If you have multiple brokers, separate `IP:port` by commas:

```
pulsar://localhost:6550,localhost:6651,localhost:6652
```

If you use TLS authentication, add `+ssl` in the scheme:

```
pulsar+ssl://pulsar.us-west.example.com:6651
```

Client

You can instantiate a PulsarClient object using just a URL for the target Pulsar cluster like this:

```
PulsarClient client = PulsarClient.builder()
        .serviceUrl("pulsar://localhost:6650")
        .build();
```

If you have multiple brokers, you can initiate a PulsarClient like this:

```
PulsarClient client = PulsarClient.builder()
        .serviceUrl("pulsar://localhost:6650,localhost:6651,localhost:6652")
        .build();
```

> **NOTE**
>
> If you run a cluster in standalone mode, the broker is available at the `pulsar://localhost:6650` URL by default.

If you create a client, you can use the `loadConf` configuration. The following parameters are available in `loadConf`.

Name	Type	Description	Default
`serviceUrl`	String	Service URL provider for Pulsar service	None
`authPluginClassName`	String	Name of the authentication plugin	None

Name	Type	Description	Default
`authParams`	String	Parameters for the authentication plugin **Example** key1:val1,key2:val2	None
`operationTimeoutMs`	long	`operationTimeoutMs`	Operation timeout
`statsIntervalSeconds`	long	Interval between each stats information Stats is activated with positive `statsInterval` Set `statsIntervalSeconds` to 1 second at least.	60
`numIoThreads`	int	The number of threads used for handling connections to brokers	1
`numListenerThreads`	int	The number of threads used for handling message listeners. The listener thread pool is shared across all the consumers and readers using the "listener" model to get messages. For a given consumer, the listener is always invoked from the same thread to ensure ordering. If you want multiple threads to process a single topic, you need to create a `shared` subscription and multiple consumers for this subscription. This does not ensure ordering.	1
`useTcpNoDelay`	boolean	Whether to use TCP no-delay flag on the connection to disable Nagle algorithm	true
`enableTls`	boolean	Whether to use TLS encryption on the connection. Note that this parameter is **deprecated**. If you want to enable TLS, use `pulsar+ssl://` in `serviceUrl` instead.	false
`tlsTrustCertsFilePath`	string	Path to the trusted TLS certificate file	None
`tlsAllowInsecureConnection`	boolean	Whether the Pulsar client accepts untrusted TLS certificate from broker	false

Name	Type	Description	Default
`tlsHostnameVerificationEnable`	boolean	Whether to enable TLS hostname verification	false
`concurrentLookupRequest`	int	The number of concurrent lookup requests allowed to send on each broker connection to prevent overload on broker	5000
`maxLookupRequest`	int	The maximum number of lookup requests allowed on each broker connection to prevent overload on broker	50000
`maxNumberOfRejectedRequestPerConnection`	int	The maximum number of rejected requests of a broker in a certain time frame (60 seconds) after the current connection is closed and the client creates a new connection to connect to a different broker	50
`keepAliveIntervalSeconds`	int	Seconds of keeping alive interval for each client broker connection	30
`connectionTimeoutMs`	int	Duration of waiting for a connection to a broker to be established If the duration passes without a response from a broker, the connection attempt is dropped	10000
`requestTimeoutMs`	int	Maximum duration for completing a request	60000
`defaultBackoffIntervalNanos`	int	Default duration for a backoff interval	TimeUnit.MILLISECONDS.toNanos(100);
`maxBackoffIntervalNanos`	long	Maximum duration for a backoff interval	TimeUnit.SECONDS.toNanos(30)

Name	Type	Description	Default
`socks5ProxyAddress`	Socket Address	SOCKS5 proxy address	None
`socks5ProxyUsername`	string	SOCKS5 proxy username	None
`socks5ProxyPassword`	string	SOCKS5 proxy password	None
`connectionMaxIdleSeconds`	int	Release the connection if it is not used for more than `connectionMaxIdleSeconds` seconds.If `connectionMaxIdleSeconds` < 0, disabled the feature that auto release the idle connection	180

Check out the Javadoc for the PulsarClient class for a full list of configurable parameters.

> In addition to client-level configuration, you can also apply producer and consumer specific configuration as described in sections below.

Client memory allocator configuration

You can set the client memory allocator configurations through Java properties.

Property	Type	Description	Default	Available values
`pulsar.allocator.pooled`	String	If set to `true`, the client uses a direct memory pool. If set to `false`, the client uses a heap memory without pool	true	• true • false
`pulsar.allocator.exit_on_oom`	String	Whether to exit the JVM when OOM happens	false	• true • false
`pulsar.allocator.leak_dete`	String	The leak detection policy for Pulsar bytebuf allocator.	Disabled	• Disabled • Simple

Property	Type	Description	Default	Available values
`ction`		• **Disabled**: No leak detection and no overhead. • **Simple**: Instruments 1% of the allocated buffer to track for leaks. • **Advanced**: Instruments 1% of the allocated buffer to track for leaks, reporting stack traces of places where the buffer is used. • **Paranoid**: Instruments 100% of the allocated buffer to track for leaks, reporting stack traces of places where the buffer is used and introduces a significant overhead.		• Advanced • Paranoid
`pulsar.allocator.out_of_memory_policy`	String	When an OOM occurs, the client throws an exception or fallbacks to heap	FallbackToHeap	• ThrowException • FallbackToHeap

Example

```
Dpulsar.allocator.pooled=true
Dpulsar.allocator.exit_on_oom=false
Dpulsar.allocator.leak_detection=Disabled
Dpulsar.allocator.out_of_memory_policy=ThrowException
```

Producer

In Pulsar, producers write messages to topics. Once you've instantiated a PulsarClient object (as in the section above), you can create a Producer for a specific Pulsar topic.

```
Producer<byte[]> producer = client.newProducer()
        .topic("my-topic")
        .create();

// You can then send messages to the broker and topic you specified:
producer.send("My message".getBytes());
```

By default, producers produce messages that consist of byte arrays. You can produce different types by specifying a message schema.

```
Producer<String> stringProducer = client.newProducer(Schema.STRING)
        .topic("my-topic")
        .create();
stringProducer.send("My message");
```

> Make sure that you close your producers, consumers, and clients when you do not need them.

```
producer.close();
consumer.close();
client.close();
```

> Close operations can also be asynchronous:

```
producer.closeAsync()
    .thenRun(() -> System.out.println("Producer closed"))
    .exceptionally((ex) -> {
        System.err.println("Failed to close producer: " + ex);
        return null;
    });
```

Configure producer

If you instantiate a `Producer` object by specifying only a topic name as the example above, the default configuration of producer is used.

If you create a producer, you can use the `loadConf` configuration. The following parameters are available in `loadConf`.

Name	Type	Description	Default
`topicName`	string	Topic name	null
`producerName`	string	Producer name	null
`sendTimeoutMs`	long	Message send timeout in ms. If a message is not acknowledged by a server before the `sendTimeout` expires, an error occurs.	30000

Name	Type	Description	Default
`blockIfQueueFull`	boolean	If it is set to `true`, when the outgoing message queue is full, the `Send` and `SendAsync` methods of producer block, rather than failing and throwing errors. If it is set to `false`, when the outgoing message queue is full, the `Send` and `SendAsync` methods of producer fail and `ProducerQueueIsFullError` exceptions occur. The `MaxPendingMessages` parameter determines the size of the outgoing message queue.	false
`maxPendingMessages`	int	The maximum size of a queue holding pending messages. For example, a message waiting to receive an acknowledgment from a broker. By default, when the queue is full, all calls to the `Send` and `SendAsync` methods fail **unless** you set `BlockIfQueueFull` to `true`.	1000
`maxPendingMessagesAcrossPartitions`	int	Deprecated, use `memoryLimit` instead. The maximum number of pending messages across partitions. Use the setting to lower the max pending messages for each partition ({@link setMaxPendingMessages(int)}) if the total number exceeds the configured value and maxPendingMessagesAcrossPartitions needs to be >= maxPendingMessages.	50000
`messageRoutingMode`	MessageRoutingMode	Message routing logic for producers on partitioned topics. Apply the logic only when setting no key on messages. Available options are as follows:	• `pulsar.RoundRobinDistribution`

Name	Type	Description	Default
		• `pulsar.RoundRobinDistribution`: round robin • `pulsar.UseSinglePartition`: publish all messages to a single partition • `pulsar.CustomPartition`: a custom partitioning scheme	
`hashingScheme`	HashingScheme	Hashing function determining the partition where you publish a particular message (**partitioned topics only**). Available options are as follows: • `pulsar.JavastringHash`: the equivalent of `string.hashCode()` in Java • `pulsar.Murmur3_32Hash`: applies the Murmur3 hashing function • `pulsar.BoostHash`: applies the hashing function from C++'s Boost library	`HashingScheme.JavastringHash`
`cryptoFailureAction`	ProducerCryptoFailureAction	Producer should take action when encryption fails. • **FAIL**: if encryption fails, unencrypted messages fail to send. • **SEND**: if encryption fails, unencrypted messages are sent.	`ProducerCryptoFailureAction.FAIL`
`batchingMaxPublishDelayMicros`	long	Batching time period of sending messages.	TimeUnit.MILLISECONDS.toMicros(1)
`batchingMaxMessages`	int	The maximum number of messages permitted in a batch.	1000
`batchingEnabled`	boolean	Enable batching of messages.	true

Name	Type	Description	Default
`chunkingEnabled`	boolean	Enable chunking of messages.	false
`compressionType`	CompressionType	Message data compression type used by a producer. Available options: • `LZ4` • `ZLIB` • `ZSTD` • `SNAPPY`	No compression
`initialSubscriptionName`	string	Use this configuration to automatically create an initial subscription when creating a topic. If this field is not set, the initial subscription is not created.	null

You can configure parameters if you do not want to use the default configuration.

For a full list, see the Javadoc for the ProducerBuilder class. The following is an example.

```
Producer<byte[]> producer = client.newProducer()
    .topic("my-topic")
    .batchingMaxPublishDelay(10, TimeUnit.MILLISECONDS)
    .sendTimeout(10, TimeUnit.SECONDS)
    .blockIfQueueFull(true)
    .create();
```

Publish to partitioned topics

By default, Pulsar topics are served by a single broker, which limits the maximum throughput of a topic. *Partitioned topics* can span multiple brokers and thus allow for higher throughput.

You can publish messages to partitioned topics using Pulsar client libraries. When publishing messages to partitioned topics, you must specify a routing mode. If you do not specify any routing mode when you create a new producer, the round-robin routing mode is used.

Routing mode

You can specify the routing mode in the `ProducerConfiguration` object used to configure your producer. The routing mode determines which partition (internal topic) each message should be

published to.

The following MessageRoutingMode options are available.

Mode	Description
`RoundRobinPartition`	If no key is provided, the producer publishes messages across all partitions in the round-robin policy to achieve the maximum throughput. Round-robin is not done per individual message. It is set to the same boundary of batching delay to ensure that batching is effective. If a key is specified on the message, the partitioned producer hashes the key and assigns the message to a particular partition. This is the default mode.
`SinglePartition`	If no key is provided, the producer picks a single partition randomly and publishes all messages into that partition. If a key is specified on the message, the partitioned producer hashes the key and assigns the message to a particular partition.
`CustomPartition`	Use custom message router implementation that is called to determine the partition for a particular message. You can create a custom routing mode by using the Java client and implementing the MessageRouter interface.

The following is an example:

```
String pulsarBrokerRootUrl = "pulsar://localhost:6650";
String topic = "persistent://my-tenant/my-namespace/my-topic";

PulsarClient pulsarClient = PulsarClient.builder().serviceUrl(pulsarBrokerRootUrl).build();
Producer<byte[]> producer = pulsarClient.newProducer()
        .topic(topic)
        .messageRoutingMode(MessageRoutingMode.SinglePartition)
        .create();
producer.send("Partitioned topic message".getBytes());
```

Custom message router

To use a custom message router, you need to provide an implementation of the MessageRouter interface, which has just one `choosePartition` method:

```
public interface MessageRouter extends Serializable {
    int choosePartition(Message msg);
}
```

The following router routes every message to partition 10:

```
public class AlwaysTenRouter implements MessageRouter {
    public int choosePartition(Message msg) {
        return 10;
    }
}
```

With that implementation, you can send messages to partitioned topics as below.

```
String pulsarBrokerRootUrl = "pulsar://localhost:6650";
String topic = "persistent://my-tenant/my-cluster-my-namespace/my-topic";

PulsarClient pulsarClient =
PulsarClient.builder().serviceUrl(pulsarBrokerRootUrl).build();
Producer<byte[]> producer = pulsarClient.newProducer()
        .topic(topic)
        .messageRouter(new AlwaysTenRouter())
        .create();
producer.send("Partitioned topic message".getBytes());
```

How to choose partitions when using a key

If a message has a key, it supersedes the round robin routing policy. The following example illustrates how to choose the partition when using a key.

```
// If the message has a key, it supersedes the round robin routing policy
        if (msg.hasKey()) {
            return signSafeMod(hash.makeHash(msg.getKey()),
topicMetadata.numPartitions());
        }

        if (isBatchingEnabled) { // if batching is enabled, choose partition on
`partitionSwitchMs` boundary.
            long currentMs = clock.millis();
            return signSafeMod(currentMs / partitionSwitchMs + startPtnIdx,
topicMetadata.numPartitions());
        } else {
            return signSafeMod(PARTITION_INDEX_UPDATER.getAndIncrement(this),
topicMetadata.numPartitions());
        }
```

Async send

You can publish messages asynchronously using the Java client. With async send, the producer puts the message in a blocking queue and returns it immediately. Then the client library sends the message to the broker in the background. If the queue is full (max size configurable), the

producer is blocked or fails immediately when calling the API, depending on arguments passed to the producer.

The following is an example.

```
producer.sendAsync("my-async-message".getBytes()).thenAccept(msgId -> {
    System.out.println("Message with ID " + msgId + " successfully sent");
});
```

As you can see from the example above, async send operations return a MessageId wrapped in a `CompletableFuture`.

Configure messages

In addition to a value, you can set additional items on a given message:

```
producer.newMessage()
    .key("my-message-key")
    .value("my-async-message".getBytes())
    .property("my-key", "my-value")
    .property("my-other-key", "my-other-value")
    .send();
```

You can terminate the builder chain with `sendAsync()` and get a future return.

Enable chunking

Message chunking enables Pulsar to process large payload messages by splitting the message into chunks at the producer side and aggregating chunked messages on the consumer side.

The message chunking feature is OFF by default. The following is an example of how to enable message chunking when creating a producer.

```
Producer<byte[]> producer = client.newProducer()
        .topic(topic)
        .enableChunking(true)
        .enableBatching(false)
        .create();
```

By default, producer chunks the large message based on max message size (`maxMessageSize`) configured at broker (eg: 5MB). However, client can also configure max chunked size using producer configuration `chunkMaxMessageSize`.

> **NOTE**

> To enable chunking, you need to disable batching (`enableBatching`=`false`) concurrently.

Intercept messages

`ProducerInterceptor` intercepts and possibly mutates messages received by the producer before they are published to the brokers.

The interface has three main events:

- `eligible` checks if the interceptor can be applied to the message.
- `beforeSend` is triggered before the producer sends the message to the broker. You can modify messages within this event.
- `onSendAcknowledgement` is triggered when the message is acknowledged by the broker or the sending failed.

To intercept messages, you can add a `ProducerInterceptor` or multiple ones when creating a `Producer` as follows.

```
Producer<byte[]> producer = client.newProducer()
        .topic(topic)
        .intercept(new ProducerInterceptor {
            @Override
            boolean eligible(Message message) {
                return true;  // process all messages
            }

            @Override
            Message beforeSend(Producer producer, Message message) {
                // user-defined processing logic
            }

            @Override
            void onSendAcknowledgement(Producer producer, Message message,
MessageId msgId, Throwable exception) {
                // user-defined processing logic
            }
        })
        .create();
```

> **NOTE**
>
> Multiple interceptors apply in the order they are passed to the `intercept` method.

Consumer

In Pulsar, consumers subscribe to topics and handle messages that producers publish to those topics. You can instantiate a new consumer by first instantiating a PulsarClient object and passing it a URL for a Pulsar broker (as above).

Once you've instantiated a PulsarClient object, you can create a Consumer by specifying a topic and a subscription.

```
Consumer consumer = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscribe();
```

The `subscribe` method will auto-subscribe the consumer to the specified topic and subscription. One way to make the consumer listen to the topic is to set up a `while` loop. In this example loop, the consumer listens for messages, prints the contents of any received message, and then acknowledges that the message has been processed. If the processing logic fails, you can use negative acknowledgment to redeliver the message later.

```
while (true) {
  // Wait for a message
  Message msg = consumer.receive();

  try {
      // Do something with the message
      System.out.println("Message received: " + new String(msg.getData()));

      // Acknowledge the message so that it can be deleted by the message broker
      consumer.acknowledge(msg);
  } catch (Exception e) {
      // Message failed to process, redeliver later
      consumer.negativeAcknowledge(msg);
  }
}
```

If you don't want to block your main thread and rather listen constantly for new messages, consider using a `MessageListener`.

```
MessageListener myMessageListener = (consumer, msg) -> {
  try {
      System.out.println("Message received: " + new String(msg.getData()));
      consumer.acknowledge(msg);
  } catch (Exception e) {
      consumer.negativeAcknowledge(msg);
  }
}

Consumer consumer = client.newConsumer()
```

```
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .messageListener(myMessageListener)
        .subscribe();
```

Configure consumer

If you instantiate a `Consumer` object by specifying only a topic and subscription name as in the example above, the consumer uses the default configuration.

When you create a consumer, you can use the `loadConf` configuration. The following parameters are available in `loadConf`.

Name	Type	Description	Default
`topicNames`	Set<String>	Topic name	Sets.newTreeSet()
`topicsPattern`	Pattern	Topic pattern	None
`subscriptionName`	String	Subscription name	None
`subscriptionType`	SubscriptionType	Subscription type Four subscription types are available: • Exclusive • Failover • Shared • Key_Shared	SubscriptionType.Exclusive
`receiverQueueSize`	int	Size of a consumer's receiver queue. For example, the number of messages accumulated by a consumer before an application calls `Receive`. A value higher than the default value increases consumer throughput, though at the expense of more memory utilization.	1000

Name	Type	Description	Default
`acknowledgementsGroupTimeMicros`	long	Group a consumer acknowledgment for a specified time. By default, a consumer uses 100ms grouping time to send out acknowledgments to a broker. Setting a group time of 0 sends out acknowledgments immediately. A longer ack group time is more efficient at the expense of a slight increase in message re-deliveries after a failure.	TimeUnit.MILLISECONDS.toMicros(100)
`negativeAckRedeliveryDelayMicros`	long	Delay to wait before redelivering messages that failed to be processed. When an application uses {@link Consumer#negativeAcknowledge(Message)}, failed messages are redelivered after a fixed timeout.	TimeUnit.MINUTES.toMicros(1)
`maxTotalReceiverQueueSizeAcrossPartitions`	int	The max total receiver queue size across partitions. This setting reduces the receiver queue size for individual partitions if the total receiver queue size exceeds this value.	50000
`consumerName`	String	Consumer name	null
`ackTimeoutMillis`	long	Timeout of unacked messages	0
`tickDurationMillis`	long	Granularity of the ack-timeout redelivery. Using an higher `tickDurationMillis` reduces the memory overhead to track	1000

Name	Type	Description	Default
		messages when setting ack-timeout to a bigger value (for example, 1 hour).	
`priorityLevel`	int	Priority level for a consumer to which a broker gives more priority while dispatching messages in Shared subscription type. It can be set at the consumer level so all topics being consumed will have the same priority level or each topic being consumed can be given a different priority level. The broker follows descending priorities. For example, 0=max-priority, 1, 2,... In Shared subscription type, the broker **first dispatches messages to the max priority level consumers if they have permits**. Otherwise, the broker considers next priority level consumers. **Example 1** If a subscription has consumerA with `priorityLevel` 0 and consumerB with `priorityLevel` 1, then the broker **only dispatches messages to consumerA until it runs out permits** and then starts dispatching messages to consumerB. **Example 2** Consumer Priority, Level, Permits C1, 0, 2 C2, 0, 1 C3, 0, 1 C4, 1, 2 C5, 1, 1 Order in which a broker dispatches messages to consumers is: C1, C2, C3, C1, C4, C5, C4.	0

Name	Type	Description	Default
`cryptoFailureAction`	ConsumerCryptoFailureAction	Consumer should take action when it receives a message that can not be decrypted. • **FAIL**: this is the default option to fail messages until crypto succeeds. • **DISCARD**:silently acknowledge and not deliver message to an application. • **CONSUME**: deliver encrypted messages to applications. It is the application's responsibility to decrypt the message. The decompression of message fails. If messages contain batch messages, a client is not be able to retrieve individual messages in batch. Delivered encrypted message contains {@link EncryptionContext} which contains encryption and compression information in it using which application can decrypt consumed message payload.	• ConsumerCryptoFailureAction.FAIL
`properties`	SortedMap<String, String>	A name or value property of this consumer. `properties` is application defined metadata attached to a consumer. When getting a topic stats, associate this metadata with the consumer stats for easier identification.	new TreeMap()
`readCompacted`	boolean	If enabling `readCompacted`, a consumer reads messages from a compacted topic rather than reading a full message backlog of a topic. A consumer only sees the latest value for each key in the compacted topic, up until	false

Name	Type	Description	Default
		reaching the point in the topic message when compacting backlog. Beyond that point, send messages as normal. Only enabling `readCompacted` on subscriptions to persistent topics, which have a single active consumer (like failure or exclusive subscriptions). Attempting to enable it on subscriptions to non-persistent topics or on shared subscriptions leads to a subscription call throwing a `PulsarClientException`.	
`subscriptionInitialPosition`	SubscriptionInitialPosition	Initial position at which to set cursor when subscribing to a topic at first time.	SubscriptionInitialPosition.Latest
`patternAutoDiscoveryPeriod`	int	Topic auto discovery period when using a pattern for topic's consumer. The default and minimum value is 1 minute.	1
`regexSubscriptionMode`	RegexSubscriptionMode	When subscribing to a topic using a regular expression, you can pick a certain type of topics. • **PersistentOnly**: only subscribe to persistent topics. • **NonPersistentOnly**: only subscribe to non-persistent topics. • **AllTopics**: subscribe to both persistent and non-persistent topics.	RegexSubscriptionMode.PersistentOnly
`deadLetterPolicy`	DeadLetterPolicy	Dead letter policy for consumers. By default, some messages are probably redelivered many times, even to the extent that it never stops.	None

Name	Type	Description	Default

		By using the dead letter mechanism, messages have the max redelivery count. **When exceeding the maximum number of redeliveries, messages are sent to the Dead Letter Topic and acknowledged automatically**.
		
		You can enable the dead letter mechanism by setting `deadLetterPolicy`.
		
		Example
		
		`client.newConsumer()`
		`.deadLetterPolicy(DeadLetterPolicy`
		`.builder().maxRedeliverCount(10).b`
		`uild())`
		`.subscribe();`
		
		Default dead letter topic name is `{TopicName}-{Subscription}-DLQ`.
		
		To set a custom dead letter topic name:
		`client.newConsumer()`
		`.deadLetterPolicy(DeadLetterPolicy`
		`.builder().maxRedeliverCount(10)`
		`.deadLetterTopic("your-topic-`
		`name").build())`
		`.subscribe();`
		
		When specifying the dead letter policy while not specifying `ackTimeoutMillis`, you can set the ack timeout to 30000 millisecond.	

`autoUpd`
`atePart`
`itions`
	boolea
	n
		If `autoUpdatePartitions` is enabled, a consumer subscribes to partition increasement automatically.
		
		Note: this is only for partitioned consumers.
			true

Name	Type	Description	Default
`replicateSubscriptionState`	boolean	If `replicateSubscriptionState` is enabled, a subscription state is replicated to geo-replicated clusters.	false
`negativeAckRedeliveryBackoff`	RedeliveryBackoff	Interface for custom message is negativeAcked policy. You can specify `RedeliveryBackoff` for a consumer.	`MultiplierRedeliveryBackoff`
`ackTimeoutRedeliveryBackoff`	RedeliveryBackoff	Interface for custom message is ackTimeout policy. You can specify `RedeliveryBackoff` for a consumer.	`MultiplierRedeliveryBackoff`
`autoAckOldestChunkedMessageOnQueueFull`	boolean	Whether to automatically acknowledge pending chunked messages when the threashold of `maxPendingChunkedMessage` is reached. If set to `false`, these messages will be redelivered by their broker.	true
`maxPendingChunkedMessage`	int	The maximum size of a queue holding pending chunked messages. When the threshold is reached, the consumer drops pending messages to optimize memory utilization.	10
`expireTimeOfIncompleteChunkedMessageMillis`	long	The time interval to expire incomplete chunks if a consumer fails to receive all the chunks in the specified time period. The default value is 1 minute.	60000
`ackReceiptEnabled`	boolean	If `ackReceiptEnabled` is enabled, ACK returns a receipt. The client got the ack receipt means the broker has processed the ack request, but if without transaction, the broker does not guarantee persistence of acknowledgments, which means the	false

Name	Type	Description	Default
		messages still have a chance to be redelivered after the broker crashes. With the transaction, the client can only get the receipt after the acknowledgments have been persistent.	

You can configure parameters if you do not want to use the default configuration. For a full list, see the Javadoc for the ConsumerBuilder class.

The following is an example.

```
Consumer consumer = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .ackTimeout(10, TimeUnit.SECONDS)
        .subscriptionType(SubscriptionType.Exclusive)
        .subscribe();
```

Async receive

The `receive` method receives messages synchronously (the consumer process is blocked until a message is available). You can also use async receive, which returns a `CompletableFuture` object immediately once a new message is available.

The following is an example.

```
CompletableFuture<Message> asyncMessage = consumer.receiveAsync();
```

Async receive operations return a Message wrapped inside of a `CompletableFuture`.

Batch receive

Use `batchReceive` to receive multiple messages for each call.

The following is an example.

```
Messages messages = consumer.batchReceive();
for (Object message : messages) {
  // do something
}
consumer.acknowledge(messages)
```

NOTE

Batch receive policy limits the number and bytes of messages in a single batch. You can specify a timeout to wait for enough messages. The batch receive is completed if any of the following conditions are met: enough number of messages, bytes of messages, wait timeout.

```
Consumer consumer = client.newConsumer()
.topic("my-topic")
.subscriptionName("my-subscription")
.batchReceivePolicy(BatchReceivePolicy.builder()
.maxNumMessages(100)
.maxNumBytes(1024 * 1024)
.timeout(200, TimeUnit.MILLISECONDS)
.build())
.subscribe();
```

The default batch receive policy is:

```
BatchReceivePolicy.builder()
.maxNumMessage(-1)
.maxNumBytes(10 * 1024 * 1024)
.timeout(100, TimeUnit.MILLISECONDS)
.build();
```

Configure chunking

You can limit the maximum number of chunked messages a consumer maintains concurrently by configuring the `maxPendingChunkedMessage` and `autoAckOldestChunkedMessageOnQueueFull` parameters. When the threshold is reached, the consumer drops pending messages by silently acknowledging them or asking the broker to redeliver them later. The `expireTimeOfIncompleteChunkedMessage` parameter decides the time interval to expire incomplete chunks if the consumer fails to receive all chunks of a message within the specified time period.

The following is an example of how to configure message chunking.

```
Consumer<byte[]> consumer = client.newConsumer()
        .topic(topic)
        .subscriptionName("test")
        .autoAckOldestChunkedMessageOnQueueFull(true)
        .maxPendingChunkedMessage(100)
        .expireTimeOfIncompleteChunkedMessage(10, TimeUnit.MINUTES)
        .subscribe();
```

Negative acknowledgment redelivery backoff

The `RedeliveryBackoff` introduces a redelivery backoff mechanism. You can achieve redelivery with different delays by setting `redeliveryCount` of messages.

```
Consumer consumer =  client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .negativeAckRedeliveryBackoff(MultiplierRedeliveryBackoff.builder()
                .minDelayMs(1000)
                .maxDelayMs(60 * 1000)
                .build())
        .subscribe();
```

Acknowledgment timeout redelivery backoff

The `RedeliveryBackoff` introduces a redelivery backoff mechanism. You can redeliver messages with different delays by setting the number of times the messages are retried.

```
Consumer consumer =  client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .ackTimeout(10, TimeUnit.SECOND)
        .ackTimeoutRedeliveryBackoff(MultiplierRedeliveryBackoff.builder()
                .minDelayMs(1000)
                .maxDelayMs(60000)
                .multiplier(2)
                .build())
        .subscribe();
```

The message redelivery behavior should be as follows.

Redelivery count	Redelivery delay
1	10 + 1 seconds
2	10 + 2 seconds
3	10 + 4 seconds
4	10 + 8 seconds
5	10 + 16 seconds

Redelivery count	Redelivery delay
6	10 + 32 seconds
7	10 + 60 seconds
8	10 + 60 seconds

> **NOTE**
>
> - The `negativeAckRedeliveryBackoff` does not work with `consumer.negativeAcknowledge(MessageId messageId)` because you are not able to get the redelivery count from the message ID.
> - If a consumer crashes, it triggers the redelivery of unacked messages. In this case, `RedeliveryBackoff` does not take effect and the messages might get redelivered earlier than the delay time from the backoff.

Multi-topic subscriptions

In addition to subscribing a consumer to a single Pulsar topic, you can also subscribe to multiple topics simultaneously using multi-topic subscriptions. To use multi-topic subscriptions you can supply either a regular expression (regex) or a `List` of topics. If you select topics via regex, all topics must be within the same Pulsar namespace.

The followings are some examples.

```
import org.apache.pulsar.client.api.Consumer;
import org.apache.pulsar.client.api.PulsarClient;

import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;

ConsumerBuilder consumerBuilder = pulsarClient.newConsumer()
        .subscriptionName(subscription);

// Subscribe to all topics in a namespace
Pattern allTopicsInNamespace = Pattern.compile("public/default/.*");
Consumer allTopicsConsumer = consumerBuilder
        .topicsPattern(allTopicsInNamespace)
        .subscribe();

// Subscribe to a subsets of topics in a namespace, based on regex
Pattern someTopicsInNamespace = Pattern.compile("public/default/foo.*");
```

```
Consumer allTopicsConsumer = consumerBuilder
        .topicsPattern(someTopicsInNamespace)
        .subscribe();
```

In the above example, the consumer subscribes to the `persistent` topics that can match the topic name pattern. If you want the consumer subscribes to all `persistent` and `non-persistent` topics that can match the topic name pattern, set `subscriptionTopicsMode` to `RegexSubscriptionMode.AllTopics`.

```
Pattern pattern = Pattern.compile("public/default/.*");
pulsarClient.newConsumer()
        .subscriptionName("my-sub")
        .topicsPattern(pattern)
        .subscriptionTopicsMode(RegexSubscriptionMode.AllTopics)
        .subscribe();
```

> **NOTE**
>
> By default, the `subscriptionTopicsMode` of the consumer is `PersistentOnly`. Available options of `subscriptionTopicsMode` are `PersistentOnly`, `NonPersistentOnly`, and `AllTopics`.

You can also subscribe to an explicit list of topics (across namespaces if you wish):

```
List<String> topics = Arrays.asList(
        "topic-1",
        "topic-2",
        "topic-3"
);

Consumer multiTopicConsumer = consumerBuilder
        .topics(topics)
        .subscribe();

// Alternatively:
Consumer multiTopicConsumer = consumerBuilder
        .topic(
            "topic-1",
            "topic-2",
            "topic-3"
        )
        .subscribe();
```

You can also subscribe to multiple topics asynchronously using the `subscribeAsync` method rather than the synchronous `subscribe` method. The following is an example.

```
Pattern allTopicsInNamespace = Pattern.compile("persistent://public/default.*");
consumerBuilder
        .topics(topics)
        .subscribeAsync()
        .thenAccept(this::receiveMessageFromConsumer);

private void receiveMessageFromConsumer(Object consumer) {
    ((Consumer)consumer).receiveAsync().thenAccept(message -> {
                // Do something with the received message
                receiveMessageFromConsumer(consumer);
            });
}
```

Subscription types

Pulsar has various subscription types to match different scenarios. A topic can have multiple subscriptions with different subscription types. However, a subscription can only have one subscription type at a time.

A subscription is identical to the subscription name; a subscription name can specify only one subscription type at a time. To change the subscription type, you should first stop all consumers of this subscription.

Different subscription types have different message distribution types. This section describes the differences between subscription types and how to use them.

To better describe their differences, assume you have a topic named "my-topic", and the producer has published 10 messages.

```
Producer<String> producer = client.newProducer(Schema.STRING)
        .topic("my-topic")
        .enableBatching(false)
        .create();
// 3 messages with "key-1", 3 messages with "key-2", 2 messages with "key-3" and 2 messages with "key-4"
producer.newMessage().key("key-1").value("message-1-1").send();
producer.newMessage().key("key-1").value("message-1-2").send();
producer.newMessage().key("key-1").value("message-1-3").send();
producer.newMessage().key("key-2").value("message-2-1").send();
producer.newMessage().key("key-2").value("message-2-2").send();
producer.newMessage().key("key-2").value("message-2-3").send();
producer.newMessage().key("key-3").value("message-3-1").send();
producer.newMessage().key("key-3").value("message-3-2").send();
producer.newMessage().key("key-4").value("message-4-1").send();
producer.newMessage().key("key-4").value("message-4-2").send();
```

Exclusive

Create a new consumer and subscribe with the `Exclusive` subscription type.

```
Consumer consumer = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Exclusive)
        .subscribe()
```

Only the first consumer is allowed to the subscription, other consumers receive an error. The first consumer receives all 10 messages, and the consuming order is the same as the producing order.

> **NOTE**
>
> If topic is a partitioned topic, the first consumer subscribes to all partitioned topics, other consumers are not assigned with partitions and receive an error.

Failover

Create new consumers and subscribe with the `Failover` subscription type.

```
Consumer consumer1 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Failover)
        .subscribe()
Consumer consumer2 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Failover)
        .subscribe()
//conumser1 is the active consumer, consumer2 is the standby consumer.
//consumer1 receives 5 messages and then crashes, consumer2 takes over as an active consumer.
```

Multiple consumers can attach to the same subscription, yet only the first consumer is active, and others are standby. When the active consumer is disconnected, messages will be dispatched to one of standby consumers, and the standby consumer then becomes the active consumer.

If the first active consumer is disconnected after receiving 5 messages, the standby consumer becomes active consumer. Consumer1 will receive:

```
("key-1", "message-1-1")
("key-1", "message-1-2")
```

```
("key-1", "message-1-3")
("key-2", "message-2-1")
("key-2", "message-2-2")
```

consumer2 will receive:

```
("key-2", "message-2-3")
("key-3", "message-3-1")
("key-3", "message-3-2")
("key-4", "message-4-1")
("key-4", "message-4-2")
```

> **NOTE**
>
> If a topic is a partitioned topic, each partition has only one active consumer, messages of one partition are distributed to only one consumer, and messages of multiple partitions are distributed to multiple consumers.

Shared

Create new consumers and subscribe with `Shared` subscription type.

```
Consumer consumer1 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Shared)
        .subscribe()

Consumer consumer2 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Shared)
        .subscribe()
//Both consumer1 and consumer2 are active consumers.
```

In Shared subscription type, multiple consumers can attach to the same subscription and messages are delivered in a round-robin distribution across consumers.

If a broker dispatches only one message at a time, consumer1 receives the following information.

```
("key-1", "message-1-1")
("key-1", "message-1-3")
("key-2", "message-2-2")
```

```
("key-3", "message-3-1")
("key-4", "message-4-1")
```

consumer2 receives the following information.

```
("key-1", "message-1-2")
("key-2", "message-2-1")
("key-2", "message-2-3")
("key-3", "message-3-2")
("key-4", "message-4-2")
```

The `Shared` subscription is different from the `Exclusive` and `Failover` subscription types. `Shared` subscription has better flexibility, but cannot provide an ordering guarantee.

Key_shared

This is a new subscription type since 2.4.0 release. Create new consumers and subscribe with `Key_Shared` subscription type.

```
Consumer consumer1 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Key_Shared)
        .subscribe()

Consumer consumer2 = client.newConsumer()
        .topic("my-topic")
        .subscriptionName("my-subscription")
        .subscriptionType(SubscriptionType.Key_Shared)
        .subscribe()
//Both consumer1 and consumer2 are active consumers.
```

Just like in the `Shared` subscription, all consumers in the `Key_Shared` subscription type can attach to the same subscription. But the `Key_Shared` subscription type is different from the `Shared` subscription. In the `Key_Shared` subscription type, messages with the same key are delivered to only one consumer in order. The possible distribution of messages between different consumers (by default we do not know in advance which keys will be assigned to a consumer, but a key will only be assigned to a consumer at the same time).

consumer1 receives the following information.

```
("key-1", "message-1-1")
("key-1", "message-1-2")
("key-1", "message-1-3")
```

```
("key-3", "message-3-1")
("key-3", "message-3-2")
```

consumer2 receives the following information.

```
("key-2", "message-2-1")
("key-2", "message-2-2")
("key-2", "message-2-3")
("key-4", "message-4-1")
("key-4", "message-4-2")
```

If batching is enabled at the producer side, messages with different keys are added to a batch by default. The broker will dispatch the batch to the consumer, so the default batch mechanism may break the Key_Shared subscription guaranteed message distribution semantics. The producer needs to use the `KeyBasedBatcher`.

```
Producer producer = client.newProducer()
        .topic("my-topic")
        .batcherBuilder(BatcherBuilder.KEY_BASED)
        .create();
```

Or the producer can disable batching.

```
Producer producer = client.newProducer()
        .topic("my-topic")
        .enableBatching(false)
        .create();
```

> **NOTE**
>
> If the message key is not specified, messages without keys are dispatched to one consumer in order by default.

Intercept messages

`ConsumerInterceptor`s intercept and possibly mutate messages received by the consumer.

The interface has six main events:

- `beforeConsume` is triggered before the message is returned by `receive()` or `receiveAsync()`. You can modify messages within this event.
- `onAcknowledge` is triggered before the consumer sends the acknowledgement to the broker.

- `onAcknowledgeCumulative` is triggered before the consumer sends the cumulative acknowledgement to the broker.
- `onNegativeAcksSend` is triggered when a redelivery from a negative acknowledgement occurs.
- `onAckTimeoutSend` is triggered when a redelivery from an acknowledgement timeout occurs.
- `onPartitionsChange` is triggered when the partitions of the (partitioned) topic change.

To intercept messages, you can add one or multiple `ConsumerInterceptor`s when creating a `Consumer` as follows.

```

 Consumer<String> consumer = client.newConsumer()
         .topic("my-topic")
         .subscriptionName("my-subscription")
         .intercept(new ConsumerInterceptor<String> {
               @Override
               public Message<String> beforeConsume(Consumer<String> consumer,
Message<String> message) {
                   // user-defined processing logic
               }

               @Override
               public void onAcknowledge(Consumer<String> consumer, MessageId
messageId, Throwable cause) {
                   // user-defined processing logic
               }

               @Override
               public void onAcknowledgeCumulative(Consumer<String> consumer,
MessageId messageId, Throwable cause) {
                   // user-defined processing logic
               }

               @Override
               public void onNegativeAcksSend(Consumer<String> consumer,
Set<MessageId> messageIds) {
                   // user-defined processing logic
               }

               @Override
               public void onAckTimeoutSend(Consumer<String> consumer,
Set<MessageId> messageIds) {
                   // user-defined processing logic
               }

               @Override
               public void onPartitionsChange(String topicName, int partitions) {
                   // user-defined processing logic
```

```
            }
        })
        .subscribe();

```

> **NOTE**
>
> If you are using multiple interceptors, they apply in the order they are passed to the `intercept` method.

Reader

With the reader interface, Pulsar clients can "manually position" themselves within a topic and read all messages from a specified message onward. The Pulsar API for Java enables you to create Reader objects by specifying a topic and a MessageId.

The following is an example.

```
byte[] msgIdBytes = // Some message ID byte array
MessageId id = MessageId.fromByteArray(msgIdBytes);
Reader reader = pulsarClient.newReader()
        .topic(topic)
        .startMessageId(id)
        .create();

while (true) {
    Message message = reader.readNext();
    // Process message
}
```

In the example above, a `Reader` object is instantiated for a specific topic and message (by ID); the reader iterates over each message in the topic after the message is identified by `msgIdBytes` (how that value is obtained depends on the application).

The code sample above shows pointing the `Reader` object to a specific message (by ID), but you can also use `MessageId.earliest` to point to the earliest available message on the topic of `MessageId.latest` to point to the most recent available message.

Configure reader

When you create a reader, you can use the `loadConf` configuration. The following parameters are available in `loadConf`.

Name	Type	Description	Default
`topicName`	String	Topic name.	None
`receiverQueueSize`	int	Size of a consumer's receiver queue. For example, the number of messages that can be accumulated by a consumer before an application calls `Receive`. A value higher than the default value increases consumer throughput, though at the expense of more memory utilization.	1000
`readerListener`	Reader Listener <T>	A listener that is called for message received.	None
`readerName`	String	Reader name.	null
`subscriptionName`	String	Subscription name	When there is a single topic, the default subscription name is `"reader-" + 10-digit UUID`. When there are multiple topics, the default subscription name is `"multiTopicsReader-" + 10-digit UUID`.
`subscriptionRolePrefix`	String	Prefix of subscription role.	null
`cryptoKeyReader`	CryptoKeyReader	Interface that abstracts the access to a key store.	null

Name	Type	Description	Default
`r`			
`cryptoFailureAction`	ConsumerCryptoFailureAction	Consumer should take action when it receives a message that can not be decrypted. • **FAIL**: this is the default option to fail messages until crypto succeeds. • **DISCARD**: silently acknowledge and not deliver message to an application. • **CONSUME**: deliver encrypted messages to applications. It is the application's responsibility to decrypt the message. The message decompression fails. If messages contain batch messages, a client is not be able to retrieve individual messages in batch. Delivered encrypted message contains {@link EncryptionContext} which contains encryption and compression information in it using which application can decrypt consumed message payload.	• ConsumerCryptoFailureAction.FAIL
`readCompacted`	boolean	If enabling `readCompacted`, a consumer reads messages from a compacted topic rather than a full message backlog of a topic. A consumer only sees the latest value for each key in the compacted topic, up until reaching the point in the topic	false

Name	Type	Description	Default
		message when compacting backlog. Beyond that point, send messages as normal. `readCompacted` can only be enabled on subscriptions to persistent topics, which have a single active consumer (for example, failure or exclusive subscriptions). Attempting to enable it on subscriptions to non-persistent topics or on shared subscriptions leads to a subscription call throwing a `PulsarClientException`.	
`resetIncludeHead`	boolean	If set to true, the first message to be returned is the one specified by `messageId`. If set to false, the first message to be returned is the one next to the message specified by `messageId`.	false

Sticky key range reader

In a sticky key range reader, broker only dispatches messages which hash of the message key contains by the specified key hash range. Multiple key hash ranges can be specified on a reader.

The following is an example to create a sticky key range reader.

```
pulsarClient.newReader()
        .topic(topic)
        .startMessageId(MessageId.earliest)
        .keyHashRange(Range.of(0, 10000), Range.of(20001, 30000))
        .create();
```

The total hash range size is 65536, so the max end of the range should be less than or equal to 65535.

Configure chunking

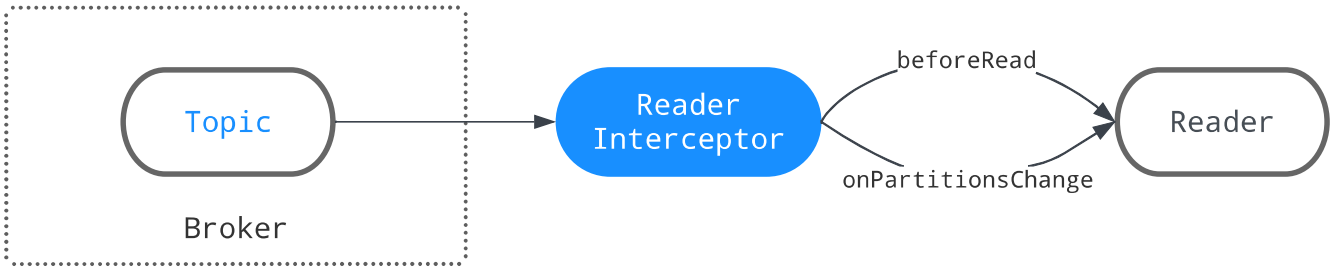
Configuring chunking for readers is similar to that for consumers. See configure chunking for consumers for more information.

The following is an example of how to configure message chunking for a reader.

```
Reader<byte[]> reader = pulsarClient.newReader()
        .topic(topicName)
        .startMessageId(MessageId.earliest)
        .maxPendingChunkedMessage(12)
        .autoAckOldestChunkedMessageOnQueueFull(true)
        .expireTimeOfIncompleteChunkedMessage(12, TimeUnit.MILLISECONDS)
        .create();
```

Create reader with interceptor

Pulsar reader interceptor intercepts and possibly mutates messages with user-defined processing before Pulsar reader reads them. With reader interceptors, you can apply unified messaging processes before messages can be read, such as modifying messages, adding properties, collecting statistics and etc, without creating similar mechanisms respectively.



Pulsar reader interceptor works on top of Pulsar consumer interceptor. The plugin interface `ReaderInterceptor` can be treated as a subset of `ConsumerInterceptor` and it has two main events.

- `beforeRead` is triggered before readers read messages. You can modify messages within this event.
- `onPartitionsChange` is triggered when changes on partitions have been detected.

To perceive triggered events and perform customized processing, you can add `ReaderInterceptor` when creating a `Reader` as follows.

```
PulsarClient pulsarClient = PulsarClient.builder().serviceUrl("pulsar://localhost:6650").build();
Reader<byte[]> reader = pulsarClient.newReader()
        .topic("t1")
        .autoUpdatePartitionsInterval(5, TimeUnit.SECONDS)
```

```
        .intercept(new ReaderInterceptor<byte[]>() {
            @Override
            public void close() {
            }

            @Override
            public Message<byte[]> beforeRead(Reader<byte[]> reader,
Message<byte[]> message) {
                // user-defined processing logic
                return message;
            }

            @Override
            public void onPartitionsChange(String topicName, int partitions) {
                // user-defined processing logic
            }
        })
        .startMessageId(MessageId.earliest)
        .create();
```

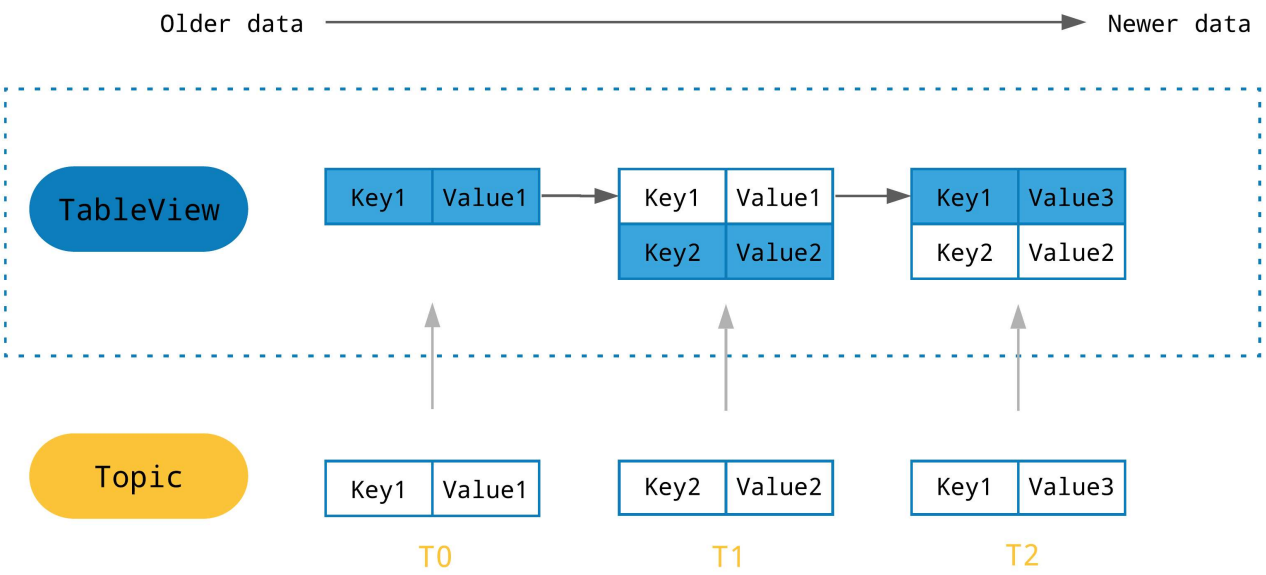
TableView

The TableView interface serves an encapsulated access pattern, providing a continuously updated key-value map view of the compacted topic data. Messages without keys will be ignored.

With TableView, Pulsar clients can fetch all the message updates from a topic and construct a map with the latest values of each key. These values can then be used to build a local cache of data. In addition, you can register consumers with the TableView by specifying a listener to perform a scan of the map and then receive notifications when new messages are received. Consequently, event handling can be triggered to serve use cases, such as event-driven applications and message monitoring.

> **NOTE**
>
> Each TableView uses one Reader instance per partition, and reads the topic starting from the compacted view by default. It is highly recommended to enable automatic compaction by configuring the topic compaction policies for the given topic or namespace. More frequent compaction results in shorter startup times because less data is replayed to reconstruct the TableView of the topic. Starting from Pulsar 2.11.0, TableView also supports reading non-persistent topics, but it does not guarantee data consistency.

The following figure illustrates the dynamic construction of a TableView updated with newer values of each key.



Configure TableView

The following is an example of how to configure a TableView.

```
TableView<String> tv = client.newTableViewBuilder(Schema.STRING)
  .topic("my-tableview")
  .create()
```

You can use the available parameters in the `loadConf` configuration or related API to customize your TableView.

Name	Type	Required?	Description	Default
`topic`	string	yes	The topic name of the TableView.	N/A
`autoUpdatePartitionInterval`	int	no	The interval to check for newly added partitions.	60 (seconds)
`subscriptionName`	string	no	The subscription name of the TableView.	null

Register listeners

You can register listeners for both existing messages on a topic and new messages coming into the topic by using `forEachAndListen`, and specify to perform operations for all existing messages by using `forEach`.

The following is an example of how to register listeners with TableView.

```
// Register listeners for all existing and incoming messages
tv.forEachAndListen((key, value) -> /*operations on all existing and incoming messages*/)

// Register action for all existing messages
tv.forEach((key, value) -> /*operations on all existing messages*/)
```

Schema

In Pulsar, all message data consists of byte arrays "under the hood." Message schemas enable you to use other types of data when constructing and handling messages (from simple types like strings to more complex, application-specific types). If you construct, say, a producer without specifying a schema, then the producer can only produce messages of type `byte[]`. The following is an example.

```
Producer<byte[]> producer = client.newProducer()
        .topic(topic)
        .create();
```

The producer above is equivalent to a `Producer<byte[]>` (in fact, you should *always* explicitly specify the type). If you'd like to use a producer for a different type of data, you need to specify a **schema** that informs Pulsar which data type will be transmitted over the topic. For more examples, see Schema - Get started.

Authentication

Pulsar Java clients currently support the following authentication mechansims:

- TLS
- JWT
- Athenz
- Kerberos
- OAuth2
- HTTP basic

Cluster-level failover

For more concepts and reference information about cluster-level failover, including concepts, benefits, use cases, constraints, usage and working principles, see Cluster-level failover.

> **TIP**

- You should configure cluster-level failover only when the cluster contains sufficient resources to handle all possible consequences. Workload intensity on the backup cluster may increase significantly.
- Connect clusters to an uninterruptible power supply (UPS) unit to reduce the risk of unexpected power loss.

Requirements

- Pulsar client 2.10 or later versions.
- For backup clusters:
  - The number of BookKeeper nodes should be equal to or greater than the ensemble quorum.
  - The number of ZooKeeper nodes should be equal to or greater than 3.
- **Turn on geo-replication** between the primary cluster and any dependent cluster (primary to backup or backup to backup) to prevent data loss.
- Set `replicateSubscriptionState` to `true` when creating consumers.

Automatic cluster-level failover | **Controlled cluster-level failover**

This is an example of how to construct a Java Pulsar client to use automatic cluster-level failover. The switchover is triggered automatically.

```
private PulsarClient getAutoFailoverClient() throws PulsarClientException {
    ServiceUrlProvider failover = AutoClusterFailover.builder()
            .primary("pulsar://localhost:6650")
            .secondary(Collections.singletonList("pulsar://other1:6650",
"pulsar://other2:6650"))
            .failoverDelay(30, TimeUnit.SECONDS)
            .switchBackDelay(60, TimeUnit.SECONDS)
            .checkInterval(1000, TimeUnit.MILLISECONDS)
            .secondaryTlsTrustCertsFilePath("/path/to/ca.cert.pem")

.secondaryAuthentication("org.apache.pulsar.client.impl.auth.AuthenticationTls",
                    "tlsCertFile:/path/to/my-
role.cert.pem,tlsKeyFile:/path/to/my-role.key-pk8.pem")

            .build();

    PulsarClient pulsarClient = PulsarClient.builder()
            .build();

    failover.initialize(pulsarClient);
```

```
18      return pulsarClient;
19  }
```

Configure the following parameters:

Parameter	Default value	Required?	Description
`primary`	N/A	Yes	Service URL of the primary cluster.
`secondary`	N/A	Yes	Service URL(s) of one or several backup clusters. You can specify several backup clusters using a comma-separated list. Note that: - The backup cluster is chosen in the sequence shown in the list. - If all backup clusters are available, the Pulsar client chooses the first backup cluster.
`failoverDelay`	N/A	Yes	The delay before the Pulsar client switches from the primary cluster to the backup cluster. Automatic failover is controlled by a probe task: 1) The probe task first checks the health status of the primary cluster. 2) If the probe task finds the continuous failure time of the primary cluster exceeds `failoverDelayMs`, it switches the Pulsar client to the backup cluster.
`switchBackDelay`	N/A	Yes	The delay before the Pulsar client switches from the backup cluster to the primary cluster. Automatic failover switchover is controlled by a probe task: 1) After the Pulsar client switches from the primary cluster to the backup cluster, the probe task continues to check the status of the primary cluster. 2) If the primary cluster functions well and

Parameter	Default value	Required?	Description
			continuously remains active longer than `switchBackDelay`, the Pulsar client switches back to the primary cluster.
`checkInterval`	30s	No	Frequency of performing a probe task (in seconds).
`secondaryTlsTrustCertsFilePath`	N/A	No	Path to the trusted TLS certificate file of the backup cluster.
`secondaryAuthentication`	N/A	No	Authentication of the backup cluster.

Edit this page