Client Libraries Node.js

The Pulsar Node.js client

You can use a Pulsar Node.js client to create producers, consumers, and readers. For Pulsar features that Python clients support, see Client Feature Matrix.

For 1.3.0 or later versions, type definitions used in TypeScript are available.

Installation

TIP

Pulsar Node.js client library is based on the C++ client library.

- You must install the Pulsar C++ client library before installing a Node.js client. For more details, see instructions.
- If an incompatible version of the C++ client is installed, you may fail to build or run the Node.js library. For the compatibility between each version of the Node.js client and the C++ client, see README.

Install the `pulsar-client` library via npm:

```
npm install pulsar-client
```

NOTE

This library only works in Node.js 10.x or later versions because it uses the `node-addon-api` module to wrap the C++ library.

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
```

Release notes

For the changelog of Pulsar Node.js clients, see release notes.

Create a client

To interact with Pulsar, you first need a client object. You can create a client instance using a `new` operator and the `Client` method, passing in a client options object (more on configuration below).

Here is an example:

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

(async () => {
  const client = new Pulsar.Client({
    serviceUrl: 'pulsar://localhost:6650',
  });

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

Client configuration

The following configurable parameters are available for Pulsar clients:

Parameter	Description	Default
`serviceUrl`	The connection URL for the Pulsar cluster. See above for more info.	
`authentication`	Configure the authentication provider. (default: no authentication). See TLS Authentication for more info.	

Parameter	Description	Default
`operationTimeoutSeconds`	The timeout for Node.js client operations (creating producers, subscribing to and unsubscribing from topics). Retries occur until this threshold is reached, at which point the operation fails.	30
`ioThreads`	The number of threads to use for handling connections to Pulsar brokers.	1
`messageListenerThreads`	The number of threads used by message listeners (consumers and readers).	1
`concurrentLookupRequest`	The number of concurrent lookup requests that can be sent on each broker connection. Setting a maximum helps to keep from overloading brokers. You should set values over the default of 50000 only if the client needs to produce and/or subscribe to thousands of Pulsar topics.	50000
`tlsTrustCertsFilePath`	The file path for the trusted TLS certificate.	
`tlsValidateHostname`	The boolean value of setup whether to enable TLS hostname verification.	`false`
`tlsAllowInsecureConnection`	The boolean value of setup whether the Pulsar client accepts untrusted TLS certificate from broker.	`false`
`statsIntervalInSeconds`	Interval between each stat info. Stats is activated with positive statsInterval. The value should be set to 1 second at least	600
`log`	A function that is used for logging.	`console.log`

Producers

Pulsar producers publish messages to Pulsar topics. You can configure Node.js producers using a producer configuration object.

Here is an example:

```
const producer = await client.createProducer({
  topic: 'my-topic', // or 'my-tenant/my-namespace/my-topic' to specify topi
tenant and namespace
});
```

```
4
5  await producer.send({
6    data: Buffer.from("Hello, Pulsar"),
7  });
8
9  await producer.close();
```

> **Promise operation**
>
> When you create a new Pulsar producer, the operation returns `Promise` object and get producer instance or an error through executor function.
> In this example, use await operator instead of executor function.

Producer operations

Pulsar Node.js producers have the following methods available:

Method	Description	Return type
`send(Object)`	Publishes a message to the producer's topic. When the message is successfully acknowledged by the Pulsar broker, or an error is thrown, the Promise object whose result is the message ID runs executor function.	`Promise<Object>`
`flush()`	Sends message from send queue to Pulsar broker. When the message is successfully acknowledged by the Pulsar broker, or an error is thrown, the Promise object runs executor function.	`Promise<null>`
`close()`	Closes the producer and releases all resources allocated to it. Once `close()` is called, no more messages are accepted from the publisher. This method returns a Promise object. It runs the executor function when all pending publish requests are persisted by Pulsar. If an error is thrown, no pending writes are retried.	`Promise<null>`
`getProducerName()`	Getter method of the producer name.	`string`
`getTopic()`	Getter method of the name of the topic.	`string`

Producer configuration

Parameter	Description	Default
`topic`	The Pulsar topic to which the producer publishes messages. The topic format is `<topic-name>` or `<tenant-name>/<namespace-name>/<topic-name>`. For example, `sample/ns1/my-topic`.	
`producerName`	A name for the producer. If you do not explicitly assign a name, Pulsar automatically generates a globally unique name. If you choose to explicitly assign a name, it needs to be unique across *all* Pulsar clusters, otherwise the creation operation throws an error.	
`sendTimeoutMs`	When publishing a message to a topic, the producer waits for an acknowledgment from the responsible Pulsar broker. If a message is not acknowledged within the threshold set by this parameter, an error is thrown. If you set `sendTimeoutMs` to -1, the timeout is set to infinity (and thus removed). Removing the send timeout is recommended when using Pulsar's message de-duplication feature.	30000
`initialSequenceId`	The initial sequence ID of the message. When producer send message, add sequence ID to message. The ID is increased each time to send.	
`maxPendingMessages`	The maximum size of the queue holding pending messages (i.e. messages waiting to receive an acknowledgment from the broker). By default, when the queue is full all calls to the `send` method fails *unless* `blockIfQueueFull` is set to `true`.	1000
`maxPendingMessagesAcrossPartitions`	The maximum size of the sum of partition's pending queue.	50000
`blockIfQueueFull`	If set to `true`, the producer's `send` method waits when the outgoing message queue is full rather than failing and throwing an error (the size of that queue is dictated by the `maxPendingMessages` parameter); if set to `false` (the default), `send` operations fails and throw a error when the queue is full.	`false`
`messageRoutingMode`	The message routing logic (for producers on partitioned topics). This logic is applied only when no key is set on messages. The available options are: round robin (`RoundRobinDistribution`), or publishing all messages to a single partition (`UseSinglePartition`, the default).	`UseSinglePartition`

Parameter	Description	Default
`hashingScheme`	The hashing function that determines the partition on which a particular message is published (partitioned topics only). The available options are: `JavaStringHash` (the equivalent of `String.hashCode()` in Java), `Murmur3_32Hash` (applies the Murmur3 hashing function), or `BoostHash` (applies the hashing function from C++'s Boost library).	`BoostHash`
`compressionType`	The message data compression type used by the producer. The available options are `LZ4`, and `Zlib`, ZSTD, SNAPPY.	Compression None
`batchingEnabled`	If set to `true`, the producer send message as batch.	`true`
`batchingMaxPublishDelayMs`	The maximum time of delay sending message in batching.	10
`batchingMaxMessages`	The maximum size of sending message in each time of batching.	1000
`properties`	The metadata of producer.	

Producer example

This example creates a Node.js producer for the `my-topic` topic and sends 10 messages to that topic:

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

(async () => {
  // Create a client
  const client = new Pulsar.Client({
    serviceUrl: 'pulsar://localhost:6650',
  });

  // Create a producer
  const producer = await client.createProducer({
    topic: 'my-topic',
  });

  // Send messages
  for (let i = 0; i < 10; i += 1) {
    const msg = `my-message-${i}`;
```

```
    producer.send({
      data: Buffer.from(msg),
    });
    console.log(`Sent message: ${msg}`);
  }
  await producer.flush();

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

Consumers

Pulsar consumers subscribe to one or more Pulsar topics and listen for incoming messages produced on that topic/those topics. You can configure Node.js consumers using a consumer configuration object.

Here is an example:

```
const consumer = await client.subscribe({
  topic: 'my-topic',
  subscription: 'my-subscription',
});

const msg = await consumer.receive();
console.log(msg.getData().toString());
consumer.acknowledge(msg);

await consumer.close();
```

> **Promise operation**
>
> When you create a new Pulsar consumer, the operation returns `Promise` object and get consumer instance or an error through executor function.
> In this example, use await operator instead of executor function.

Consumer operations

Pulsar Node.js consumers have the following methods available:

Method	Description	Return type
`receive()`	Receives a single message from the topic. When the message is available, the Promise object run executor function and get message	`Prc Obj`

Method	Description	Return type
	object.	
`receive(Number)`	Receives a single message from the topic with specific timeout in milliseconds.	`Promise<Object>`
`acknowledge(Object)`	Acknowledges a message to the Pulsar broker by message object.	`void`
`acknowledgeId(Object)`	Acknowledges a message to the Pulsar broker by message ID object.	`void`
`acknowledgeCumulative(Object)`	Acknowledges *all* the messages in the stream, up to and including the specified message. The `acknowledgeCumulative` method returns void, and send the ack to the broker asynchronously. After that, the messages are *not* redelivered to the consumer. Cumulative acking can not be used with a shared subscription type.	`void`
`acknowledgeCumulativeId(Object)`	Acknowledges *all* the messages in the stream, up to and including the specified message ID.	`void`
`negativeAcknowledge(Message)`	Negatively acknowledges a message to the Pulsar broker by message object.	`void`
`negativeAcknowledgeId(MessageId)`	Negatively acknowledges a message to the Pulsar broker by message ID object.	`void`
`close()`	Closes the consumer, disabling its ability to receive messages from the broker.	`Promise<null>`
`unsubscribe()`	Unsubscribes the subscription.	`Promise<null>`

Consumer configuration

Parameter	Description	Default
`topic`	The Pulsar topic on which the consumer establishes a subscription and listen for messages.	
`topics`	The array of topics.	
`topicsPattern`	The regular expression for topics.	
`subscription`	The subscription name for this consumer.	
`subscriptionType`	Available options are `Exclusive`, `Shared`, `Key_Shared`, and `Failover`.	`Exclusive`
`subscriptionInitialPosition`	Initial position at which to set cursor when subscribing to a topic at first time.	`SubscriptionInitialPosition.Latest`
`ackTimeoutMs`	Acknowledge timeout in milliseconds.	0
`nAckRedeliverTimeoutMs`	Delay to wait before redelivering messages that failed to be processed.	60000
`receiverQueueSize`	Sets the size of the consumer's receiver queue, i.e. the number of messages that can be accumulated by the consumer before the application calls `receive`. A value higher than the default of 1000 could increase consumer throughput, though at the expense of more memory utilization.	1000
`receiverQueueSizeAcrossPartitions`	Set the max total receiver queue size across partitions. This setting is used to reduce the receiver queue size for individual partitions if the total exceeds this value.	50000
`consumerName`	The name of consumer. Currently(v2.4.1), failover mode use consumer name in ordering.	
`properties`	The metadata of consumer.	
`listener`	A listener that is called for a message received.	
`readCompacted`	If enabling `readCompacted`, a consumer reads messages from a compacted topic rather than reading a full message backlog of a	false

Parameter	Description	Default
	topic. A consumer only sees the latest value for each key in the compacted topic, up until reaching the point in the topic 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 (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`.	

Consumer example

This example creates a Node.js consumer with the `my-subscription` subscription on the `my-topic` topic, receives messages, prints the content that arrive, and acknowledges each message to the Pulsar broker for 10 times:

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

(async () => {
  // Create a client
  const client = new Pulsar.Client({
    serviceUrl: 'pulsar://localhost:6650',
  });

  // Create a consumer
  const consumer = await client.subscribe({
    topic: 'my-topic',
    subscription: 'my-subscription',
    subscriptionType: 'Exclusive',
  });

  // Receive messages
  for (let i = 0; i < 10; i += 1) {
    const msg = await consumer.receive();
    console.log(msg.getData().toString());
    consumer.acknowledge(msg);
  }

  await consumer.close();
```

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

Instead, a consumer can be created with `listener` to process messages.

```
// Create a consumer
const consumer = await client.subscribe({
  topic: 'my-topic',
  subscription: 'my-subscription',
  subscriptionType: 'Exclusive',
  listener: (msg, msgConsumer) => {
    console.log(msg.getData().toString());
    msgConsumer.acknowledge(msg);
  },
});
```

> **NOTE**
>
> Pulsar Node.js client uses AsyncWorker. Asynchronous operations such as creating consumers/producers and receiving/sending messages are performed in worker threads. Until completion of these operations, worker threads are blocked. Since there are only 4 worker threads by default, a called method may never be complete. To avoid this situation, you can set `UV_THREADPOOL_SIZE` to increase the number of worker threads, or define `listener` instead of calling `receive()` many times.

Readers

Pulsar readers process messages from Pulsar topics. Readers are different from consumers because with readers you need to explicitly specify which message in the stream you want to begin with (consumers, on the other hand, automatically begin with the most recently unacked message). You can configure Node.js readers using a reader configuration object.

Here is an example:

```
const reader = await client.createReader({
  topic: 'my-topic',
  startMessageId: Pulsar.MessageId.earliest(),
});

const msg = await reader.readNext();
console.log(msg.getData().toString());

await reader.close();
```

Reader operations

Pulsar Node.js readers have the following methods available:

Method	Description	Return type
`readNext()`	Receives the next message on the topic (analogous to the `receive` method for consumers). When the message is available, the Promise object run executor function and get message object.	`Promise<Object>`
`readNext(Number)`	Receives a single message from the topic with specific timeout in milliseconds.	`Promise<Object>`
`hasNext()`	Return whether the broker has next message in target topic.	`Boolean`
`close()`	Closes the reader, disabling its ability to receive messages from the broker.	`Promise<null>`

Reader configuration

Parameter	Description	Default
`topic`	The Pulsar topic on which the reader establishes a subscription and listen for messages.	
`startMessageId`	The initial reader position, i.e. the message at which the reader begins processing messages. The options are `Pulsar.MessageId.earliest` (the earliest available message on the topic), `Pulsar.MessageId.latest` (the latest available message on the topic), or a message ID object for a position that is not earliest or latest.	
`receiverQueueSize`	Sets the size of the reader's receiver queue, i.e. the number of messages that can be accumulated by the reader before the application calls `readNext`. A value higher than the default of 1000 could increase reader throughput, though at the expense of more memory utilization.	1000
`readerName`	The name of the reader.	
`subscriptionRolePrefix`	The subscription role prefix.	

Parameter	Description	Default
`readCompacted`	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 reaching the point in the topic 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 (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`.	`false`

Reader example

This example creates a Node.js reader with the `my-topic` topic, reads messages, and prints the content that arrive for 10 times:

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

(async () => {
  // Create a client
  const client = new Pulsar.Client({
    serviceUrl: 'pulsar://localhost:6650',
    operationTimeoutSeconds: 30,
  });

  // Create a reader
  const reader = await client.createReader({
    topic: 'my-topic',
    startMessageId: Pulsar.MessageId.earliest(),
  });

  // read messages
  for (let i = 0; i < 10; i += 1) {
    const msg = await reader.readNext();
    console.log(msg.getData().toString());
  }

  await reader.close();
```

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

Messages

In Pulsar Node.js client, you have to construct producer message objects for producers.

Here is an example of a message:

```
const msg = {
  data: Buffer.from('Hello, Pulsar'),
  partitionKey: 'key1',
  properties: {
    'foo': 'bar',
  },
  eventTimestamp: Date.now(),
  replicationClusters: [
    'cluster1',
    'cluster2',
  ],
}

await producer.send(msg);
```

The following keys are available for producer message objects:

Parameter	Description
`data`	The actual data payload of the message.
`properties`	A Object for any application-specific metadata attached to the message.
`eventTimestamp`	The timestamp associated with the message.
`sequenceId`	The sequence ID of the message.
`partitionKey`	The optional key associated with the message (particularly useful for things like topic compaction).
`replicationClusters`	The clusters to which this message is replicated. Pulsar brokers handle message replication automatically; you should only change this setting if you want to override the broker default.

Parameter	Description
`deliverAt`	The absolute timestamp at or after which the message is delivered.
`deliverAfter`	The relative delay after which the message is delivered.

Message object operations

In Pulsar Node.js client, you can receive (or read) message objects as consumers (or readers).

The message object has the following methods available:

Method	Description	Return type
`getTopicName()`	Getter method of topic name.	`String`
`getProperties()`	Getter method of properties.	`Array<Object>`
`getData()`	Getter method of message data.	`Buffer`
`getMessageId()`	Getter method of message id object.	`Object`
`getPublishTimestamp()`	Getter method of publish timestamp.	`Number`
`getEventTimestamp()`	Getter method of event timestamp.	`Number`
`getRedeliveryCount()`	Getter method of redelivery count.	`Number`
`getPartitionKey()`	Getter method of partition key.	`String`

Message ID object operations

In Pulsar Node.js client, you can get message id objects from message objects.

The message id object has the following methods available:

Method	Description	Return type
`serialize()`	Serialize the message id into a Buffer for storing.	`Buffer`

Method	Description	Return type
`toString()`	Get message id as String.	`String`

The client has a static method of message id object. You can access it as `Pulsar.MessageId.someStaticMethod`.

The following static methods are available for the message id object:

Method	Description	Return type
`earliest()`	MessageId representing the earliest, or oldest available message stored in the topic.	`Object`
`latest()`	MessageId representing the latest, or last published message in the topic.	`Object`
`deserialize(Buffer)`	Deserialize a message id object from a Buffer.	`Object`

End-to-end encryption

Pulsar encryption allows applications to encrypt messages at producers and decrypt messages at consumers. See Get started for more details.

Edit this page