# Confluent JavaScript Client for Apache Kafka

Confluent, a leading developer and maintainer of Apache Kafka®, offers the Confluent JavaScript Client for Apache Kafka®. This client integrates JavaScript applications with Kafka clusters and enables you to produce and consume messages from Kafka topics.

The JavaScript Client is a wrapper around the librdkafka C library, supporting various Kafka features with high performance and reliability.

There are two variants of the API offered by this client library: the promisified API and the callback-based API. Here are some recommendations on how to choose:

- If you're starting a new deployment, use the promisified API.
- If you're migrating from KafkaJS, use the KafkaJS migration guide to get started.
- If you're migrating from node-rdkafka, use the node-rdkafka migration guide to get started.

Regardless of which API you choose, this document, along with the API reference and the examples, serves as a comprehensive guide to using the JavaScript Client.

## Installation

**Requirements**

- The following configurations are supported for the JavaScript Client:
  - Supported versions of Node.js: The two LTS versions, 18 and 20, and the latest versions, 21 and 22.
  - Linux (x64 and arm64): Both glibc and musl/alpine.
  - macOS: arm64/m1. macOS (Intel) is supported on a best-effort basis.
  - Windows: x64.

  Installation on any of the above platforms is seamlessly supported and does not require C/C++ compilation.
- To install the JavaScript Client on an unsupported system that is not listed above, a supported version of Python must be available on the system for the installation process. This is required for the node-gyp build tool.
- Yarn and pnpm (Performant Node Package Manager) support is experimental.

**Install on supported environments**

To install the JavaScript Client, run the following command:

```
npm install @confluentinc/kafka-javascript
```

**Install on unsupported environments**

To install the client an OS or an architecture that isn't supported by the prebuilt binaries:

1. Install a recent version of Python, a C++ compiler that supports C++20 or above. It's best to use a recent version of `gcc` or `clang`.
2. Install librdkafka. There are many ways to do this depending on your target machine. The easiest way is to grab it from your package manager, or if not available, you might need to build it from the source.

   Example installation for Debian/Ubuntu based platforms:

   ```
   sudo mkdir -p /etc/apt/keyrings
   wget -qO - https://packages.confluent.io/deb/7.8/archive.key | gpg --dearmor | sudo tee /etc/apt/keyrings/confluent.gpg > /dev/
   null
   sudo apt-get update
   sudo apt install librdkafka-dev
   ```

It is recommended that you follow detailed, up to date instructions available on the librdkafka repository.

3. Once librdkafka is installed, set the environment variables. An example of this for a Unix-like platform is shown below:

```
export CKJS_LINKING=dynamic
export BUILD_LIBRDKAFKA=0
```

4. Run `npm install` to build the library and link it against the librdkafka installed in the system:

```
npm install @confluentinc/kafka-javascript
```

This process sometimes requires troubleshooting. If you encounter any issues, please search for similar issues in the repository, or file one.

If `npm install` takes more than a couple of minutes and you're using a MacOS, you can check this issue. for information.

## Promisified API

The promisified is included in your code as:

```
const {
  Kafka,
  ErrorCodes, CompressionTypes, // You can specify additional optional properties for further configuration.
} = require('@confluentinc/kafka-javascript').KafkaJS;
```

### Configure

JavaScript Client supports many librdkafka configuration options.

Configuration keys that have the suffix `_cb` are designated as callbacks. Currently, the only supported callbacks are the `rebalance_cb` for the consumer, and `oauthbearer_token_refresh_cb` for all clients.

In addition, a `kafkaJS` configuration key can be used to specify an object whose keys can be a limited set of KafkaJS properties. This should only be used if you're migrating from KafkaJS.

### Produce

A `Producer` sends messages to Kafka. The following example illustrates how to create an instance and send a message:

```
const producer = new Kafka().producer({
    'bootstrap.servers': '<fill>',
});

await producer.connect();

const deliveryReports = await producer.send({
    topic: 'test-topic',
    messages: [
      { value: 'v1', key: 'x' },
    ]
});

console.log({deliveryReports});
await producer.disconnect();
```

The `send` method is used to send messages to the Kafka cluster. It returns an array of delivery reports for each message sent. The producer must be connected to use this method.

Multiple messages can be sent to a topic in a single `send`. Besides the value, a message can contain several other fields. For example:

```
const message = {
  // An optional key for the message. If a key is provided, all messages with the same key will be sent to the same partition. It can be a string, a Buffer, or null.
  key: null,
  // The value of the message. This can be a Buffer or a string, and must be set.
  value: Buffer.from('Hello'),
  // The partition to send the message to. It may be a number, or be unset. If not set, the partition will be chosen by the partitioner based on the key.
  partition: 1,
  // Headers to be sent along with the message. May be unset.
  headers: {
    // Header keys are strings.
    // Header values can be strings.
    'header-key-0': 'header-value',
    // Header values can be buffers, too.
    'header-key-1': Buffer.from('header-value'),
    // One or more values can be sent for a single key.
    'header-key-3': [Buffer.from('header-value-0'), Buffer.from('header-value-1')],
  }
  // The timestamp of the message in milliseconds since the Unix epoch. If not set, the broker will set the timestamp.
  timestamp?: string
}
```

## Message batching, async sends, and flushing

Messages are batched internally by the library on a per partition basis.

Setting the `linger.ms` property allows the library to wait up to the specified amount of time to accumulate messages into a single batch.

The maximum size of the batch is determined by `batch.num.messages` or `batch.size`.

Note that if `send` is awaited on, *only* the messages included within that `send` can be batched together.

For the best possible throughput, it's recommended to use `send` calls without awaiting them, and then await all of them together. Additionally, `linger.ms` can be increased to accumulate a larger number of messages into the same batch to reduce network and CPU overhead.

```
const reportPromises = [];
for (const topicMessages of topicsMessages) {
  reportPromises.push(
    producer.send({topic: topicMessages.topic, messages: topicMessages.messages}));
}

await reportPromises;
```

On the other hand, if you want to ensure that the latency of each message is as low as possible, you can await each `send` individually and set `linger.ms` to 0 or 1.

```
const producer = new Kafka().producer({
    'bootstrap.servers': '<fill>',
    'linger.ms': 1,
});

await producer.connect();
await producer.send(/* ... */);
```

In case you are producing messages with a larger `linger.ms`, and want to send messages immediately, you can call `flush` to send all messages in the internal buffer without waiting for the `linger.ms` to expire.

```
 // Unawaited send.
const reportPromise = producer.send(/*...*/);

// Try to send everything, waiting upto 1s. If we weren't able to finish sending in 1s, throw an error.
await producer.flush({ timeout: 1000 });
```

Warning

If you increase the value of `linger.ms`, do not await your `send` calls, as every single await may take up to `linger.ms` to resolve. Instead, if you need the delivery report, flush after sending as shown above.

For further details, see compression and retries/reliability.

## Producer methods

| Method | Description |
|---|---|
| async connect() | Connects to the broker. Must be called exactly once before using the producer |
| async disconnect() | Disconnects producer (flushing it) and cleans up connections and resources. |
| async send() | Sends message(s) to a Kafka topic. Returns the delivery reports on awaiting. |

| Method | Description |
|---|---|
| async sendBatch() | Sends message(s) to various Kafka topics. Returns the delivery reports on awaiting. |
| async flush() | Flushes any messages buffered in the library and attempts to send them immediately. |
| async transaction() | Starts a new transaction. See Transactional producer and exactly-once semantics. |
| async sendOffsets() | Commits offsets within a transaction. See Transactional producer and exactly-once semantics. |
| async commit() | Commits an ongoing transaction. See Transactional producer and exactly-once semantics. |
| async abort() | Aborts an ongoing transaction. See Transactional producer and exactly-once semantics. |
| dependentAdmin() | Returns an Admin client which uses this producer for its underlying connections. |

| Method | Description |
|---|---|
| setSaslCredentials() | Sets the SASL username and password to be used for the next login attempt. Used for SASL PLAIN or SASL SCRAM authentication. |

See the JavaScript Client API reference documentation for more details.

## Consume

A `Consumer` receives messages from Kafka. The following example illustrates how to create an instance and start consuming messages:

```
const consumer = new Kafka().consumer({
    'bootstrap.servers': '<fill>',
    'group.id': 'test-group', // Mandatory property for a consumer - the consumer group id.
});

await consumer.connect();
await consumer.subscribe({ topics: ["test-topic"] });

consumer.run({
  eachMessage: async ({ topic, partition, message }) => {
    console.log({
      topic,
      partition,
      headers: message.headers,
      offset: message.offset,
      key: message.key?.toString(),
      value: message.value.toString(),
    });
  }
});

// Whenever we're done consuming, maybe after user input or a signal:
await consumer.disconnect();
```

The consumer must be connected before calling `run`. The `run` method starts the consumer loop, which takes care of polling the cluster, and will call the `eachMessage` callback for every message you get from the cluster.

The message may contain several other fields besides the value. For example:

```
{
  // Key of the message - may not be set.
  key: Buffer.from('key'),
  // Value of the message - will be set.
  value: Buffer.from('value'),
  // The timestamp set by the producer or the broker in milliseconds since the Unix epoch.
  timestamp: '1734008723000',
  // The current epoch of the leader for this partition.
  leaderEpoch: 2,
  // Size of the message in bytes.
  size: 6,
  // Offset of the message on the partition.
  offset: '42',
  // Headers that were sent along with the message.
  headers: {
    'header-key-0': ['header-value-0', 'header-value-1'],
    'header-key-1': Buffer.from('header-value'),
  }
}
```

A message is considered to be processed successfully when `eachMessage` for that message runs to completion without throwing an error. In case an error is thrown, the message is marked unprocessed, and `eachMessage` will be called with the same message again.

### Subscribe and rebalance

To consume messages, the consumer must be a part of a consumer group, and it must subscribe to one or more topics. The group is specified with the `group.id` property, and the `subscribe` method should be called after connecting to the cluster.

The consumer does not actually join the consumer group until `run` is called. Joining a consumer group causes a rebalance within all the members of that consumer group, where each consumer is assigned a set of partitions to consume from. Rebalances may also be caused when a consumer leaves a group by disconnecting, or when new partitions are added to a topic.

It is possible to add a callback to track rebalances:

```
const rebalance_cb = (err, assignment) => {
  switch (err.code) {
    case ErrorCodes.ERR__ASSIGN_PARTITIONS:
      console.log(`Assigned partitions ${JSON.stringify(assignment)}`);
      break;
    case ErrorCodes.ERR__REVOKE_PARTITIONS:
      console.log(`Revoked partitions ${JSON.stringify(assignment)}`);
      break;
    default:
      console.error(err);
  }
};
const consumer = new Kafka().consumer({
    'bootstrap.servers': '<fill>',
    'group.id': 'test-group',
    'rebalance_cb': rebalance_cb,
});
```

It's also possible to modify the assignment of partitions, or pause consumption of newly assigned partitions just after a rebalance.

```
const rebalance_cb = (err, assignment, assignmentFns) => {
  switch (err.code) {
    case ErrorCodes.ERR__ASSIGN_PARTITIONS:
      // Change the assignment as needed - this mostly boils down to changing the offset to start consumption from, though
      // you are free to do anything.
      if (assignment.length > 0) {
        assignment[0].offset = 34;
      }
      assignmentFns.assign(assignment);
      // Can pause consumption of new partitions just after a rebalance.
      break;
    case ErrorCodes.ERR__REVOKE_PARTITIONS:
      break;
    default:
      console.error(err);
  }
};
```

Subscriptions can be changed anytime, and the running consumer triggers a rebalance whenever that happens. The current assignment of partitions to the consumer can be checked with the `assign` method.

### Commit offsets

As a part of a consumer group, the consumer can commit offsets to the broker, so any consumer from the same group can pick up from where it left off in case of a crash or a restart. By default, offsets are committed automatically by the library.

By default, every 5000ms, the last successfully processed offset (that is, for which `eachMessage` ran to completion) for each partition is committed to the broker. Disconnecting the consumer or a rebalance also causes offsets to be committed.

It's possible to control this interval using the property `auto.commit.interval.ms`.

It’s possible to commit offsets manually, as well, by setting `enable.auto.commit` to `false`:

```
const consumer = new Kafka().consumer({
    'bootstrap.servers': '<fill>',
    'group.id': 'test-group',
    'enable.auto.commit': false,
});

consumer.run({
  eachMessage: async ({ topic, partition, message }) => {
    console.log(message);
    if (isItTimeToCommit) {
      await consumer.commitOffsets([
        {
          topic,
          partition,
          // Note how we are committing offset + 1, and not offset. This is
          // Kafka expects the offsets as 'the next offset to read', not 'the last offset processed'.
          offset: (parseInt(message.offset, 10) + 1).toString(),
        },
      ]);
    }
  },
});

// commitOffsets can also be called without an argument to commit all the successfully
// processed offsets upto that point in time.
setInterval(() => {
  consumer.commitOffsets();
}, 1000);
```

Warning

If you are manually handling offsets, make sure to set up a `rebalance_cb` and commit offsets in case of `ERR__REVOKE_PARTITIONS`, as the library will not commit offsets automatically even in that case.

## Pause and seek

It’s possible to pause consumption of owned partitions. The library will not make any more requests to the broker to fetch messages for that partition until it’s resumed.

```
// Pause topic, partitions 0 and 1.
const resumeMethod = consumer.pause([{ topic, partitions: [0, 1] }]);

// Resume - after a while.
resumeMethod();

// Can also resume manually specifying partitions.
consumer.resume([{ topic, partitions: [1] }]);
```

Partitions not in the assignment cannot be paused (the pause method doesn’t have an effect).

It’s possible to seek to specific offsets of a partition. This is useful for reprocessing or skipping messages.

```
consumer.run({
  eachMessage: async ({ topic, partition, message }) => {
    if (restartFromEarliest) {
      // offset can be a positive number (in a string), or be '-2' representing the 'earliest' offset available on the broker.
      consumer.seek({topic, partition, offset: '-2'});
    }
  },
});
```

If a partition belongs to the assignment, it's "seeked" immediately. If not, the seek is persisted in the library, and if that partition is assigned to the consumer at a later point through a rebalance, it's "seeked" then.

If `enable.auto.commit` is `true` (it is by default), then seeking to an offset also causes a commit to the broker with that offset.

## Advanced consumer options

The consumer can consume messages concurrently. If your message processing involves async operations, you can pass a number to the `partitionsConsumedConcurrently` option to the `run` method. The library will attempt to dispatch up to that number of `eachMessage` at the same time. Note that, for a single partition, the order of messages is guaranteed, and thus no more than one message from a partition will trigger `eachMessage` at the same time.

Messages can also be consumed in batches. The `eachBatch` callback can be provided to the `run` method instead of the `eachMessage` callback. Currently, the batch size has a fixed upper limit of 32. The batches passed to the method do not correspond to the actual batches of messages on the broker, but logical batches which the library creates by accumulating messages from the broker.

Since there is no way for the library to tell when individual messages in a batch are processed, the library will commit offsets for the entire batch when the `eachBatch` callback runs to completion. If an error is thrown from the callback, no offsets are committed, and the callback is dispatched again for the same batch.

For finer-grained control, the argument passed to `eachBatch` contains a `resolve` method. Calling the method allows the library to commit offsets for the messages processed up to that point, even if there's an error in processing further messages within the batch. It's also possible to turn off auto-resolution completely, and only use the `resolve` method, by passing `eachBatchAutoResolve: false` to the `run` method. `partitionsConsumedConcurrently` is also applicable to `eachBatch`.

```
consumer.run({
    partitionsConsumedConcurrently,
    eachBatchAutoResolve: isAutoResolve,
    eachBatch: async event => {
        console.log(event.batch.messages);

        // If we're not auto-resolving, we need to resolve the offsets manually.
        // Note that unlike commit, we don't need to add 1 here to the message offset.
        if (!isAutoResolve)
            event.resolveOffset(event.batch.messages[event.batch.messages.length - 1].message.offset);
    }
});
```

## Consumer methods

| Method | Description |
|---|---|
| async connect() | Connects to the broker. Must be called exactly once before using the consumer |
| async disconnect() | Disconnects consumer, triggering one final rebalance, and cleans up connections and resources. |

| Method | Description |
|---|---|
| async subscribe() | Subscribes consumer to one or more topics. May add to or replace current subscription. |
| async run() | Starts the main consume loop, polling the broker, and triggering message process callbacks. |
| assignment() | Gets the current assignment of partitions for this consumer. |
| async commitOffsets() | Commits offsets to the broker, up to the last processed offset, or custom ranges specified in the argument. |
| async committed() | Fetches committed offsets for the assigned topic partitions, or a custom topic partition list. |
| pause() | Pauses consumption for a subset of assigned partitions. |
| resume() | Resumes consumption for paused partitions. |
| paused() | Allows checking which partitions are currently paused. |

| Method | Description |
|---|---|
| seek() | Allows seeking to a specific offset in assigned partitions (immediately) or unassigned partitions (to be “seeked” when assigned). |
| dependentAdmin() | Returns an Admin client which uses this producer for its underlying connections. |

See the JavaScript Client API reference documentation for more details.

## Transactional producer and exactly-once semantics

The JavaScript Client library supports idempotent producers, transactional producers, and exactly-once semantics (EOS).

To use an idempotent producer:

```
const producer = new Kafka().producer({
    'bootstrap.servers': '<fill>',
    'enable.idempotence': true,
});
```

More details about the guarantees provided by an idempotent producer can be found here, as well as the limitations and other configuration changes that an idempotent producer brings.

To use a transactional producer:

```
const producer = new Kafka().producer({
    'bootstrap.servers': '<fill>',
    'transactional.id': 'my-transactional-id', // Must be unique for each producer instance.
});

await producer.connect();

// Start transaction.
await producer.transaction();
await producer.send({topic: 'topic', messages: [{value: 'message'}]});

// Commit transaction.
await producer.commit();
```

Specifying a `transactional.id` makes the producer transactional. The `transactional.id` must be unique for each producer instance. The producer must be connected before starting a transaction with `transaction()`. A transactional producer cannot be used as a non-transactional producer, and every message must be within a transaction.

More details about the guarantees provided by a transactional producer can be found here.

Using a transactional producer also allows for exactly-once semantics (EOS) in the specific case of consuming from a Kafka cluster, processing the message, and producing to another topic on the same cluster.

```
consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
        try {
            const transaction = await producer.transaction();
            await transaction.send({
                topic: 'produceTopic',
                messages: [
                    { value: 'consumed a message: ' + message.value.toString() },
                ]
            });

            await transaction.sendOffsets({
                consumer,
                topics: [
                    {
                        topic,
                        partitions: [
                            { partition, offset: String(Number(message.offset) + 1) },
                        ],
                    }
                ],
            });

            // The transaction assures that the message sent and the offset committed
            // are transactional, only reflecting on the broker on commit.
            await transaction.commit();
        } catch (e) {
            console.error(e);
            await transaction.abort();
        }
    },
});
```

## Admin client

The JavaScript Client library includes an admin client to interact with the Kafka cluster. The admin client provides several methods to manage topics, groups, and other Kafka entities.

```
// An admin client can be created from configuration.
const admin = new Kafka().admin({
    'bootstrap.servers': '<fill>',
});

// Or from a producer or consumer instance.
const depAdmin = producer.dependentAdmin();

await admin.connect();
await depAdmin.connect();
```

A complete list of methods available on the admin client can be found in the JavaScript Client API reference documentation.

## Error handling

Methods from the library throw `KafkaJSError` when they encounter a problem. Each error object contains a `code` property, which should be compared against the `ErrorCodes` object exported by the library to decide how to handle the error.

```
const { Kafka, ErrorCodes, isKafkaJSError } = require('@confluentinc/kafka-javascript').KafkaJS;
try {
  await producer.send({ /*... */ });
} catch (e) {
  if (!isKafkaJSError()) throw e;
  if (err.code === ErrorCodes.ERR__MSG_TIMED_OUT) {
    // Some backoff and retry logic.
  }
}
```

After the client has successfully connected, the client stays useable despite any errors that it throws. The client will attempt to reconnect to the broker in case of a disconnection, and will continue to work as expected if the cluster is reachable. It is not recommended to terminate clients on errors indiscriminately, rather, it's better to check the error and take specific action.

There is a `fatal` attribute on the error. In very rare cases (relating to idempotent or transactional producers, and static consumer group membership, both of which are properties you would have configured yourself), the error may be fatal, and the client becomes unusable. In such cases, it's recommended to terminate the client and create a new one.

The error object also contains a `retriable` and `abortable` properties which are set only while using a transactional producer. The former indicates that the failed operation may be retried, and the latter indicates that the transaction should be aborted.

## Logging

The library provides a built-in logger, and it allows you to set a custom logger.

The default logger logs to `console`.

```
const { logLevel } = require('@confluentinc/kafka-javascript').KafkaJS;

// You can use the built-in logger.
const libraryLogger = producer.logger();
libraryLogger.info('This is an info message');

// And modify the log level on the fly.
libraryLogger.setLogLevel(logLevel.INFO);
```

Most of the internal workings of the library are logged by librdkafka at the `loglevel.DEBUG`. It uses component-wise logging system.

To turn on debug logs, set the `debug` property in the configuration object to a comma-separated list of components you want to debug, or `all` to turn on all debug logs.

A list of available components can be found here. Setting `debug` in the configuration also changes the in-built logger to start with a default log level of `logLevel.DEBUG`.

Note

Because the `debug` configuration cannot be changed on the fly, a majority of the debug logs are not available in the library if you don't set the `debug` property before connecting. Therefore, using `libraryLogger.setLogLevel(logLevel.DEBUG)` is generally not very useful to debug library internals. However, going from `logLevel.DEBUG` to `logLevel.INFO` or `logLevel.ERROR` is still useful.

A custom logger may also be provided, as long as it conforms to the following interface:

```
class MyLogger {
  constructor(); // You can freely change the signature of the constructor.
  setLogLevel(logLevel); // logLevel is a member of the logLevel enum.
  info(message, extra); // message is a string, extra is an object with additional information.
  error(message, extra);
  warn(message, extra);
  debug(message, extra);
}

// Usage
const customLogger = new MyLogger();
const producer = new Kafka().producer({
    'bootstrap.servers': '<fill>',
    'debug': 'all',
    "kafkaJS": {
        "logger": customLogger,
    }
});
```

## Callback-based API

The callback-based API is included in your code as:

```
const Kafka = require('@confluentinc/kafka-javascript');
```

### Configure

The JavaScript Client supports many librdkafka Configuration options.

If you pass in a unsupported configuration option, an error will be thrown.

Configuration keys that have the suffix `_cb` are designated as callbacks:

- Some of these keys are informational and you can choose to opt-in (for example, `dr_cb`).
- Others are callbacks designed to return a value, such as `partitioner_cb`.

The JavaScript Client library currently supports the following callbacks:

- `partitioner_cb`
- `dr_cb` or `dr_msg_cb`
- `event_cb`
- `rebalance_cb` (see Rebalancing)
- `offset_commit_cb` (see Commits)
- `oauthbearer_token_refresh_cb`

### librdkafka methods

The JavaScript Client library includes two utility functions for detecting the status of your installation. Where applicable, try to include these functions when reporting issues:

- Get the features supported by your compilation of the librdkafka library by reading the variable `features` on the root of the `confluent-kafka-javascript` object:

  ```
  const Kafka = require('@confluentinc/kafka-javascript');
  console.log(Kafka.features);

  // #=> [ 'gzip', 'snappy', 'ssl', 'sasl', 'regex', 'lz4' ]
  ```

- Get the version of the librdkafka:

  ```
  const Kafka = require('@confluentinc/kafka-javascript');
  console.log(Kafka.librdkafkaVersion);

  // #=> 2.3.0
  ```

## Produce

### Initialization

The `Producer` constructor takes a configuration object to send messages to Kafka. For example:

```
const producer = new Kafka.Producer({
  'bootstrap.servers': 'kafka-host1:9092,kafka-host2:9092'
});
```

A `Producer` only requires `bootstrap.servers` (the Kafka brokers) to be created. The values in this list are separated by commas. For other configuration options, see the librdkafka Configuration options.

The following example illustrates a list with several librdkafka options set.

```
const producer = new Kafka.Producer({
  'client.id': 'kafka',
  'bootstrap.servers': 'localhost:9092',
  'compression.codec': 'gzip',
  'retry.backoff.ms': 200,
  'message.send.max.retries': 10,
  'socket.keepalive.enable': true,
  'queue.buffering.max.messages': 100000,
  'queue.buffering.max.ms': 1000,
  'batch.num.messages': 1000000,
  'dr_cb': true
});
```

### Stream API

You can use the `Producer` as a writable stream immediately after creation. For example:

```
// Our producer with its Kafka brokers
// This call returns a new writable stream to our topic 'topic-name'
const stream = Kafka.Producer.createWriteStream({
  'bootstrap.servers': 'kafka-host1:9092,kafka-host2:9092'
}, {}, {
  topic: 'topic-name',
  autoClose: false,
  pollInterval: 1000,
});

// Writes a message to the stream
const queuedSuccess = stream.write(Buffer.from('Awesome message'));

if (queuedSuccess) {
  // This only indicates that the message was queued, not sent to Kafka.
  console.log('We queued our message!');
} else {
  console.log('Too many messages in our queue already');
}

// NOTE: MAKE SURE TO LISTEN TO THIS IF YOU WANT THE STREAM TO BE DURABLE
// Otherwise, any error will bubble up as an uncaught exception.
stream.on('error', (err) => {
  // Here's where we'll know if something went wrong sending to Kafka
  console.error('Error in our producer stream');
  console.error(err);
})
```

Note

If you do not want your code to crash when an error happens, ensure you have an `error` listener on the stream. Most errors are not necessarily fatal, but the ones that are will immediately destroy the stream. If you use `autoClose`, the stream will close itself at the first sign of a problem.

## Standard API

The Standard API is more performant, particularly when handling high volumes of messages. However, it requires more manual setup to use. The following example illustrates its use:

```
const producer = new Kafka.Producer({
  'bootstrap.servers': 'localhost:9092',
  'dr_cb': true
});

// Connect to the broker manually
producer.connect();

// Wait for the ready event before proceeding
producer.on('ready', () => {
  try {
    producer.produce(
      // Topic to send the message to
      'topic',
      // optionally we can manually specify a partition for the message
      // this defaults to -1 - which will use librdkafka's default partitioner (consistent random for keyed messages, random for un
keyed messages)
      null,
      // Message to send. Must be a buffer
      Buffer.from('Awesome message'),
      // for keyed messages, we also specify the key - note that this field is optional
      'Stormwind',
      // you can send a timestamp here. If your broker version supports it,
      // it will get added. Otherwise, we default to 0
      Date.now(),
      // you can send an opaque token here, which gets passed along
      // to your delivery reports
    );
  } catch (err) {
    console.error('A problem occurred when sending our message');
    console.error(err);
  }
});

// Any errors we encounter, including connection errors
producer.on('event.error', (err) => {
  console.error('Error from producer');
  console.error(err);
})

// We must either call .poll() manually after sending messages
// or set the producer to poll on an interval (.setPollInterval).
// Without this, we do not get delivery events and the queue
// will eventually fill up.
producer.setPollInterval(100);

// You can also set up the producer to poll in the background thread which is
// spawned by the C code. It is more efficient for high-throughput producers.
// Calling this clears any interval set in setPollInterval.
producer.setPollInBackground(true);
```

To see the configuration options available to you, see the librdkafka Configuration options.

## Producer methods

| Method | Description |
|---|---|
| connect() | Connects to the broker. The `connect()` method emits the `ready` event when it connects successfully. If it does not, the error will be passed through the callback. |

| Method | Description |
|---|---|
| disconnect() | Disconnects from the broker. The `disconnect()` method emits the `disconnected` event when it has disconnected. If it does not, the error will be passed through the callback. |
| poll() | Polls the producer for delivery reports or other events to be transmitted via the `emitter`. In order to get the events in the librdkafka queue to emit, you must call this regularly or use one of the methods which automatically poll the producer for you. |
| setPollInterval(interval) | Polls the producer on this interval, handling disconnections and reconnection. Set it to `0` to turn it off. |
| setPollInBackground(set) | Polls the producer on the librdkafka background thread. Pass a truthy value to turn it on, a non-truthy value to turn it off. |
| produce(topic, partition, msg, key, timestamp, opaque) | Sends a message. The `produce()` method throws when produce would return an error. Ordinarily, this is just if the queue is full. |
| flush(timeout, callback) | Flushes the librdkafka internal queue, sending all messages. The default timeout is `500ms`. |
| initTransactions(timeout, callback) | Initializes the transactional producer. |
| beginTransaction(callback) | Starts a new transaction. |
| sendOffsetsToTransaction(offsets, consumer, timeout, callback) | Sends consumed `topic-partition-offsets` to the broker, which will get committed along with the transaction. |
| abortTransaction(timeout, callback) | Aborts the ongoing transaction. |
| commitTransaction(timeout, callback) | Commits the ongoing transaction. |

| Method | Description |
|---|---|
| setSaslCredentials(username, password) | Change SASL credentials to be sent on the next authentication attempt. Only applicable if SASL authentication is being used. |

## Events

Some configuration properties that end in `_cb` indicate that an event should be generated for that option. You have two options:

- Provide a value of `true` and react to the event
- Provide a callback function directly

The following example illustrates an event:

```
const producer = new Kafka.Producer({
  'client.id': 'my-client', // Specifies an identifier to use to help trace activity in Kafka
  'bootstrap.servers': 'localhost:9092', // Connect to a Kafka instance on localhost
  'dr_cb': true // Specifies that we want a delivery-report event to be generated
});

// Poll for events every 100 ms
producer.setPollInterval(100);

producer.on('delivery-report', (err, report) => {
  // Report of delivery statistics here:
  //
  console.log(report);
});
```

The following table describes types of events.

| Event | Description |
|---|---|
| disconnected | The `disconnected` event is emitted when the broker has disconnected. This event is emitted only when `.disconnect` is called. The wrapper will always try to reconnect otherwise. |
| ready | The `ready` event is emitted when the Producer is ready to send messages. |
| event | The `event` event is emitted when librdkafka reports an event (if you opted in via the `event_cb` option). |
| event.log | The `event.log` event is emitted when logging events come in (if you opted into logging via the `event_cb` option). You will need to set a value for `debug` if you want to send information. |
| event.stats | The `event.stats` event is emitted when librdkafka reports `stats` (if you opted in by setting the `statistics.interval.m` to a non-zero value). |
| event.error | The `event.error` event is emitted when librdkafka reports an error. |

| Event | Description |
|---|---|
| event.throttle | The `event.throttle` event emitted when librdkafka reports throttling. |
| delivery-report | The `delivery-report` event is emitted when a delivery report has been found via polling. To use this event, you must set `request.required.acks` to `1` or `-1` in topic configuration and `dr_cb` (or `dr_msg_cb` if you want the report to contain the message payload) to `true` in the `Producer` constructor options. |

### Higher level producer

The higher level producer is a variant of the producer which can propagate callbacks to you upon message delivery. The rest of the class behavior is the same as the producer class.

```
const producer = new Kafka.HighLevelProducer({
  'bootstrap.servers': 'localhost:9092',
});
```

This will enrich the produce call so it will have a callback to tell you when the message has been delivered. You lose the ability to specify opaque tokens.

```
producer.produce(topicName, null, Buffer.from('alliance4ever'), null, Date.now(), (err, offset) => {
  // The offset if our acknowledgment level allows us to receive delivery offsets
  console.log(offset);
});
```

Additionally, you can add serializers to modify the value of a produce for a key or value before it is sent over to Kafka.

```
producer.setValueSerializer((value) => {
  return Buffer.from(JSON.stringify(value));
});

producer.setTopicValueSerializer((topic, value) => {
  // Process value based on topic it's being produced to.
  return processedValue;
});
```

Otherwise the behaviour of the class is exactly the same.

## Consume

### Initialization

To read messages from Kafka, instantiate a `KafkaConsumer` object as follows:

```
const consumer = new Kafka.KafkaConsumer({
  'group.id': 'kafka',
  'bootstrap.servers': 'localhost:9092',
}, {});
```

- The first parameter is the global config.

  The `group.id` and `bootstrap.servers` properties are required for a consumer.

- The second parameter is an optional topic configuration that gets applied to all subscribed topics.

To view a list of all supported configuration properties, see librdkafka Configurations. Look for the `C` and `*` keys.

### Rebalancing

By default, rebalancing is managed internally by librdkafka. To override this functionality, you may provide your own logic as a rebalance callback.

```
const consumer = new Kafka.KafkaConsumer({
  'group.id': 'kafka',
  'bootstrap.servers': 'localhost:9092',
  'rebalance_cb': (err, assignment) => {

    if (err.code === ErrorCodes.ERR__ASSIGN_PARTITIONS) {
      // Note: this can throw when you are disconnected. Take care and wrap it in
      // a try catch if that matters to you
      this.assign(assignment);
    } else if (err.code == ErrorCodes.ERR__REVOKE_PARTITIONS){
      // Same as above
      this.unassign();
    } else {
      // We had a real error
      console.error(err);
    }

  }
})
```

`this` is bound to the `KafkaConsumer` you have created. By specifying a `rebalance_cb` you can also listen to the `rebalance` event as an emitted event. This event is not emitted when using the internal librdkafka rebalancer.

### Commits

When you commit in `confluent-kafka-javascript`, the standard way is to queue the commit request up with the next librdkafka request to the broker. When doing this, there isn’t a way to know the result of the commit.

To get the result of the commit, you can use another callback you can listen to.

```
const consumer = new Kafka.KafkaConsumer({
  'group.id': 'kafka',
  'bootstrap.servers': 'localhost:9092',
  'offset_commit_cb': (err, topicPartitions) => {

    if (err) {
      // There was an error committing
      console.error(err);
    } else {
      // Commit went through. Let's log the topic partitions
      console.log(topicPartitions);
    }

  }
})
```

`this` is bound to the `KafkaConsumer` you have created. By specifying an `offset_commit_cb` you can also listen to the `offset.commit` event as an emitted event. It receives an error and the list of topic partitions as argument. This is not emitted unless opted in.

## Message structure

Messages that are returned by the `KafkaConsumer` have the following structure:

```
{
  value:     // message contents as a Buffer,
  size:      // size of the message, in bytes,
  topic:     // topic the message comes from,
  offset:    // offset the message was read from,
  partition: // partition the message was on,
  key:       // key of the message if present,
  timestamp: // timestamp of message creation if present
}
```

For example:

```
{
  value: Buffer.from('hi'),
  size: 2,
  topic: 'librdtesting-01',
  offset: 1337,
  partition: 1,
  key: 'someKey',
  timestamp: 1510325354780
}
```

## Stream API

The stream API is the easiest way to consume messages. The following example illustrates the use of the stream API:

```
// Read from the librdtesting-01 topic... Note that this creates a new stream on each call, so call it once and store it
const stream = KafkaConsumer.createReadStream(globalConfig, topicConfig, {
  topics: ['librdtesting-01']
});

stream.on('data', (message) => {
  console.log('Got message');
  console.log(message.value.toString());
});
```

You can also get the `consumer` from the streamConsumer, for using consumer methods. The following example:

```
stream.consumer.commit(); // Commits all locally stored offsets
```

## Standard API

You can use the Standard API and manage callbacks and events yourself.

The two modes for consuming messages are supported: flowing mode and non-flowing mode.

- *Flowing mode*. This mode flows all of the messages it can read by maintaining an infinite loop in a separate thread. It only stops when it detects the consumer has issued the *unsubscribe* or *disconnect* method. Messages are sent to you either with a callback that you provide to the *consume* method, or by listening to the *data* event.

  The following example illustrates the flowing mode:

```
// Flowing mode
consumer.connect();

consumer
  .on('ready', () => {
    consumer.subscribe(['librdtesting-01']);

    // Consume from the librdtesting-01 topic. This is what determines
    // the mode we are running in.
    // In case consume() is called without an argument, or with only one argument
    // which is the callback function, it works in the flowing mode.

    consumer.consume();
  })
  .on('data', (data) => {
    // Output the actual message contents
    console.log(data.value.toString());
  });
```

- *Non-flowing mode*. This mode reads up to a specific number of messages from Kafka at once, and does not create a separate thread for fetching. It only stops when it detects the consumer has issued the `unsubscribe` or `disconnect` method.

  The following example illustrates the non-flowing mode:

```
// Non-flowing mode
consumer.connect();

consumer
  .on('ready', () => {
    // Subscribe to the librdtesting-01 topic
    // This makes subsequent consumes read from that topic.
    consumer.subscribe(['librdtesting-01']);

    // Read up to 1000 message every 1000 milliseconds
    setInterval(() => {
      consumer.consume(1);
    }, 1000);
  })
  .on('data', (data) => {
    console.log('Message found!  Contents below.');
    console.log(data.value.toString());
  });
```

## Consumer methods

| Method | Description |
|---|---|
| consumer.connect() | Connects to the broker. The `connect()` emits the event `ready` when it has successfully connected. If it does not, the error will be passed through the callback. |
| consumer.disconnect() | Disconnects from the broker. The `disconnect()` method emits `disconnected` when it has disconnected. If it does not, the error will be passed through the callback. |
| consumer.subscribe(topics) | Subscribes to an array of topics. |
| consumer.unsubscribe() | Unsubscribes from the currently subscribed topics. You cannot subscribe to different topics without calling the `unsubscribe()` method first. |

| Method | Description |
|---|---|
| consumer.consume(cb?) | Gets messages from the existing subscription as quickly as possible. If `cb` is specified, invokes `cb(err, message)`. This method keeps a background thread running to do the work. Note that the number of worker threads in a NodeJS process is limited by `UV_THREADPOOL_SIZE` (default value is `4`) and using up all of them blocks other parts of the application that need threads. If you need multiple consumers then consider increasing `UV_THREADPOOL_SIZE` or using `consumer.consume(number, cb)` instead. |
| consumer.consume(number, cb?) | Gets `number` of messages from the existing subscription. If `cb` is specified, invokes `cb(err, message)`. |
| consumer.commit() | Commits all locally stored offsets. |
| consumer.commit(topicPartition) | Commits offsets specified by the topic partition. |
| consumer.commitMessage(message) | Commits the offsets specified by the message. |

## Events

| Event | Description |
|---|---|
| data | When using the Standard API consumed messages are emitted in this event. |
| partition.eof | When using Standard API and the configuration option `enable.partition.eof` is set, `partition.eof` events are emitted in this event. The event contains `topic`, `partition`, and `offset` properties. |
| warning | The event is emitted in case of `UNKNOWN_TOPIC_OR_PART` or `TOPIC_AUTHORIZATION_FAILED` errors when consuming in flowing mode. Since the consumer will continue working if the error is still happening, the warning event should reappear after the next metadata refresh. To control the metadata refresh rate set the `topic.metadata.refresh.interval.ms` property. Once you resolve the error, you can manually call `getMetadata` to speed up consumer recovery. |
| disconnected | The `disconnected` event is emitted when the broker disconnects. This event is only emitted when `.disconnect` is called. The wrapper will always try to reconnect otherwise. |
| ready | The `ready` event is emitted when the Consumer is ready to read messages. |
| event | The `event` event is emitted when librdkafka reports an event (if you opted in via the `event_cb` option). |
| event.log | The `event.log` event is emitted when logging events occur (if you opted in for logging via the `event_cb` option). You will need to set a value for `debug` if you want information to send. |

| Event | Description |
|---|---|
| event.stats | The `event.stats` event is emitted when librdkafka reports stats (if you opted in by setting the `statistics.interval.ms` to a non-zero value). |
| event.error | The `event.error` event is emitted when librdkafka reports an error. |
| event.throttle | The `event.throttle` event is emitted when librdkafka reports throttling. |

## Read current offsets from the broker for a topic

To query the latest (and earliest) offset for one of your topics, use `queryWaterMarkOffsets` with connected producers and consumers both allow:

```
const timeout = 5000, partition = 0;
consumer.queryWatermarkOffsets('my-topic', partition, timeout, (err, offsets) => {
  const high = offsets.highOffset;
  const low = offsets.lowOffset;
});

producer.queryWatermarkOffsets('my-topic', partition, timeout, (err, offsets) => {
  const high = offsets.highOffset;
  const low = offsets.lowOffset;
});
```

An error will return if the client is not connected or the request timed out within the specified interval.

## Metadata

To retrieve metadata from Kafka, use the `getMetadata` method with `Kafka.Producer` or `Kafka.KafkaConsumer`.

When fetching metadata for a specific topic, if a topic reference does not exist, one is created using the default configuration.

See the documentation on `Client.getMetadata` if you want to set configuration parameters, for example, `acks` on a topic to produce messages to.

The following example illustrates how to use the `getMetadata` method.

```
const opts = {
  topic: 'librdtesting-01',
  timeout: 10000
};

producer.getMetadata(opts, (err, metadata) => {
  if (err) {
    console.error('Error getting metadata');
    console.error(err);
  } else {
    console.log('Got metadata');
    console.log(metadata);
  }
});
```

Metadata on any connection is returned in the following data structure:

```
{
  orig_broker_id: 1,
  orig_broker_name: "broker_name",
  brokers: [
    {
      id: 1,
      host: 'localhost',
      port: 40
    }
  ],
  topics: [
    {
      name: 'awesome-topic',
      partitions: [
        {
          id: 1,
          leader: 20,
          replicas: [1, 2],
          isrs: [1, 2]
        }
      ]
    }
  ]
}
```

### Admin client

The library provides an admin client to interact with the Kafka cluster. The admin client provides several methods to manage topics, groups, and other Kafka entities.

To following code snippet instantiates the `AdminClient`:

```
const Kafka = require('@confluentinc/kafka-javascript');

const client = Kafka.AdminClient.create({
  'client.id': 'kafka-admin',
  'bootstrap.servers': 'broker01'
});

// From an existing producer or consumer
const depClient = Kafka.AdminClient.createFrom(producer);
```

These will instantiate and connect the `AdminClient`, which will allow the calling of the admin methods.

A complete list of methods available on the admin client can be found in the API reference documentation.

## OAuthbearer callback authentication

The JavaScript Client library supports OAuthBearer token authentication for both the promisified and the callback-based API. The token is fetched using a callback provided by the user. The callback is called at 80% of the token expiry time, and the library uses the new token for the next login attempt.

```
async function token_refresh(oauthbearer_config /* string - passed from config */, cb /* can be used if function is not async */) {
    // Some logic to fetch the token, before returning it.
    return { tokenValue, lifetime, principal, extensions };
}

const producer = new Kafka().producer({
    'bootstrap.servers': '<fill>',
    'security.protocol': 'sasl_ssl', // or sasl_plain
    'sasl.mechanisms': 'OAUTHBEARER',
    'sasl.oauthbearer.config': 'someConfigPropertiesKey=value', // Just passed straight to token_refresh as a string, carries no ot
her significance.
    'oauthbearer_token_refresh_cb': token_refresh,
});
```

For a special case of OAuthBearer token authentication, where the token is fetched from an OIDC provider using the `client_credentials` grant type, the library provides a built-in callback, which can be set through just the configuration without any custom function required:

```
const producer = new Kafka().producer({
    'bootstrap.servers': '<fill>',
    'security.protocol': 'sasl_ssl', // or sasl_plain
    'sasl.mechanisms': 'OAUTHBEARER',
    'sasl.oauthbearer.method': 'oidc',
    'sasl.oauthbearer.token.endpoint.url': issuerEndpointUrl,
    'sasl.oauthbearer.scope': scope,
    'sasl.oauthbearer.client.id': oauthClientId,
    'sasl.oauthbearer.client.secret': oauthClientSecret,
    'sasl.oauthbearer.extensions': `logicalCluster=${kafkaLogicalCluster},identityPoolId=${identityPoolId}`
});
```

These examples are for the promisified API, but the callback-based API can be used with the same configuration settings.