



English ▼



Overview ▼

Processing streaming data

Parsing responses

The body of a streaming API response consists of a series of newline-delimited messages, where “newline” is considered to be `\r\n` (in hex, `0x0D 0x0A`) and “message” is a [JSON](#) encoded data structure or a blank line.

Note that Tweet content may sometimes contain linefeed `\n` characters, but will not contain carriage returns `\r`. Therefore, to make sure you get whole message payloads, break out each message on `\r\n` boundaries, as `\n` may occur in the middle of a message. Alternatively, use the delimited parameter explained under [delimited messages](#).

See [streaming message types](#) for information about the message formats you will receive from the streaming API.

JSON data

The individual messages streamed by this API are [JSON encoded](#). Keep in mind that the attributes of a JSON-encoded object are unordered - do not rely on fields appearing in any given order. Also keep in mind that your JSON parser should tolerate unexpected or missing fields.

Missing counts

In very rare cases the Streaming API may elect to deliver an incomplete Tweet field instead of waiting for data which is taking a long time to fetch. In the case of a numeric count, this will manifest as a -1 value. If you happen to see any counts with values of -1, use the REST API to backfill accurate values as needed.

```
"user":{ "followers_count":-1, "friends_count":-1, "listed_count":null, "created_at":"We
```

Transfer-Encoding: chunked

Most streaming connections will be encoded using [chunked transfer encoding](#), as indicated by the presence of a `Transfer-Encoding: chunked` HTTP header in the response. Because most HTTP libraries deal with chunked transfer encoding transparently, this document will assume that your code has access to the reassembled HTTP stream and does not have to deal with this encoding.

In the off-chance that your code is actually parsing the raw TCP stream, you will need to reassemble HTTP chunks manually. Be aware that Tweets and other streamed messages will not necessarily fall on HTTP chunk boundaries. If you use the [delimited parameter](#) you will receive both transfer encoding lengths (HTTP level) and delimited Tweet lengths (Application level).

Delimited messages

By passing `delimited=length` when connecting to a stream (note that the value is the literal string `length`, not a number) each message will be preceded by a string representation of a base-10 integer indicating the length of the message in bytes. Note that this is independent of, and does not affect any chunked transfer encoding. Clients may use these length delimiters to more efficiently copy chunks of text off of the incoming stream, rather than having to parse message text for `\r\n` tokens.

To illustrate how this may be implemented, here is some pseudocode which reads length-delimited messages from a stream:

```
while (true) {  
  do {  
    lengthBytes = readline()  
  } while (lengthBytes.length < 1)  
  messageLength = parseInt(lengthBytes);  
  messageBytes = read(messageLength);  
  enqueueForMarkupProcessor(messageBytes);  
}
```

Falling behind

Clients which are unable to process messages fast enough will be disconnected. A way to track whether your client is falling behind is to compare the timestamp of the Tweets you receive with the current time. If the difference between the timestamps increases over time, then the client is not processing Tweets as fast as they are being delivered. Another way to receive notifications that a client is falling behind is to pass the [stall_warnings](#) parameter when

establishing the streaming connection.

Scaling

Twitter streaming volume is not constant. Throughout the course of a 24 hour period, there is a natural ebb and flow to the number of Tweets delivered per second. In addition, the amount of data grows steadily month-over-month, and significant world or cultural events may cause traffic spikes of 3 or more times the current daily peak volume. Clients which want to maintain their connections must provision and test for these cases.

The best practice for ingesting Tweets and other streaming messages is to decouple collection and processing of high volume streams. For example, collect the raw text of messages in one process, passing each message into a message queue, rotated flatfile, or database. A second process or set of processes should parse the messages and extract any necessary fields for storage or further manipulation.

Message ordering

Messages from the Streaming API are not delivered in sorted order. Instead, they are usually delivered within a few seconds of a total ordering. On occasion, a small proportion may be delivered tens of seconds to several minutes out of order.

A totally ordered timeline display may cause a user to miss some messages as out of order messages are inserted further down the timeline. Consider sorting a timeline only when out of focus and appending to the top when in focus.

Delete messages may be delivered before the original Tweet so implementations should be able to replay a local cache of unrecognized deletes.

Duplicate messages

Duplicate messages may be delivered so implementations should be tolerant of receiving a Tweet more than once. This is most likely to occur when reconnecting to or backfilling from the Streaming API. Backfilling from the REST API will also tend to produce duplicates. Make sure to handle duplicate messages and use the `since_id` REST parameter to reduce duplication, latency and server-side load.

Gzip compression

Gzip compression may reduce the bandwidth needed to process a stream to as small as 1/5th

the size of an uncompressed stream. Request a gzipped stream by connecting with the following HTTP header:

```
Accept-Encoding: deflate, gzip
```

Twitter will respond with a gzipped stream.

Note: There are cases where Twitter will not return a compressed stream, even if one was requested. Always check the `Content-Encoding` header to verify that the stream is actually being compressed. To make sure you get a compressed stream:

- Make a HTTP 1.1 request.
- Include a `User-Agent` header. Any value should be fine.
- Include a valid `Host` header.
- Do **not** send a `Connection: close` header.

Gzip and EventMachine

The Ruby [EventMachine library](#) defaults to sending a `Connection: close` header, which will suppress gzip encoding. To prevent this, pass `:keepalive => true` when connecting to the streaming endpoint. EventMachine currently only supports deflate compressed streams, so send a `Accept-Encoding: deflate` header.

An example request line for an EventMachine integration:

```
@ http = EventMachine::HttpRequest.new('STREAMING URL').post(:body=>BODY, :head => {"Cor
```

Gzip and Java


Java clients which use `java.util.zip.GZIPInputStream()` and wrap it with a `java.io.BufferedReader()` to read streaming API data will encounter buffering on low volume streams, since `GZIPInputStream`'s `available()` method is not suitable for streaming purposes. To fix this, create a subclass of `GZIPInputStream()` which overrides the `available()` method. For example:

```
import java.io.IOException;
import java.io.InputStream;
import java.util.zip.GZIPInputStream;

final class StreamingGZIPInputStream extends GZIPInputStream {
    private final InputStream wrapped;
```

```
public StreamingGZIPInputStream(InputStream is) throws IOException {
    super(is);
    wrapped = is;
}

/**
 * Overrides behavior of GZIPInputStream which assumes we have all the data available
 * which is not true for streaming. We instead rely on the underlying stream to tell
 * how much data is available.
 *
 * Programs should not count on this method to return the actual number
 * of bytes that could be read without blocking.
 *
 * @return - whatever the wrapped InputStream returns
 * @exception IOException if an I/O error occurs.
 */
public int available() throws IOException {
    return wrapped.available();
}
}
```



To use this class, replace the use of GZIPInputStream, as in this example:

```
Reader reader = new BufferedReader(new InputStreamReader(new GZIPInputStream(http.getIn
```



With StreamingGZipInputStream:

```
Reader reader = new BufferedReader(new InputStreamReader(new StreamingGZIPInputStream(ht
```



SOLUTIONS

[Customer ServiceDigs](#)[Build Great Apps Sign in with](#)[Tell Great Stories Twitter](#)[Twitter Social](#)[Crashlytics](#)[Answers by](#)[Crashlytics](#)[Beta by](#)[Crashlytics](#)[MoPub](#)

FABRIC

DATA & WEB

[Gnip](#)[REST APIs](#)[Streaming APIs](#)[Twitter for](#)[Websites](#)

RESOURCES

[Documentation](#)[Forums](#)[Blog](#)[Case Studies](#)[API Terms](#)[Policy Support](#)

PROGRAMS

[Official Partner](#)[Events](#)[Flight 2014](#)

TOOLS

[API Status](#)[API Console](#)[Manage Your](#)[Apps](#)[Cards Validator](#)[Subscribe to Our Newsletter](#)

Copyright © 2016 Twitter, Inc.

All Rights Reserved

[About](#) [Company](#) [Blog](#) [Help](#) [Status](#) [Jobs](#) [Terms](#) [Privacy](#) [Cookies](#) [Ads info](#) [Brand](#)
[Advertise](#) [Businesses](#) [Developers](#)

© 2016 Twitter, Inc.