

---

# Streaming API Developer Guide

Version 63.0, Spring '25





# CONTENTS

<b>Chapter 1: Getting Started with Streaming API</b>	<b>1</b>
Push Technology	2
Bayeux Protocol, CometD, and Long Polling	2
Streaming API Terms	2
How the Client Connects	3
Message Reliability	7
Message Durability	7
Streaming Event Features	11
Streaming API vs. Pub/Sub API	13
Differences Between Change Events Received with Streaming API vs. Pub/Sub API	13
API End-of-Life Policy	16
<b>Chapter 2: Code Examples</b>	<b>17</b>
Example: Subscribe to and Replay Events Using a Java Client (EMP Connector)	18
Prerequisites	19
Step 1: Create an Object	19
Step 2: Create a PushTopic (Legacy)	20
Step 3: Download and Build the Project	21
Step 4: Use the Connector with Username and Password Login	22
(Optional) Step 5: Use the Connector with OAuth Bearer Token Login	24
Learn More About EMP Connector	25
Example: Subscribe to and Replay Events Using a Lightning Component	27
Example: Subscribe to and Replay Events Using a Visualforce Page	28
Prerequisites	28
Deploy a Sample Project to Your Org	28
Assign a Permission Set	30
Durable PushTopic Streaming Sample	30
Durable Generic Streaming Sample	33
Replay Events Sample: Code Walkthrough	37
Example: Interactive Visualforce Page without Replay	39
Prerequisites	40
Step 1: Create an Object	40
Step 2: Create a PushTopic	41
Step 3: Create the Static Resources	42
Step 4: Create a Visualforce Page	43
Step 5: Test the PushTopic Channel	44
Example: Authentication	45
Set Up Authentication for Developer Testing	45
Set Up Authorization with OAuth 2.0	45


<b>Chapter 3: Client Connection Considerations</b>	<b>49</b>
Clients and Timeouts	50
Clients and Cookies for Streaming API	50
Supported CometD Versions	50
Streaming API Calls Blocked by Security Settings	51
Debugging Streaming API Applications	51
Handling Streaming API Errors	51
Streaming API Error Codes	54
Using an Experience Cloud Site with Streaming API-Based Features	57
<b>Chapter 4: PushTopic Events (Legacy)</b>	<b>58</b>
Working with PushTopics	59
PushTopic Queries	59
Event Notification Rules	65
Replay PushTopic Streaming Events	71
Filtered Subscriptions	71
Bulk Subscriptions	72
Deactivating a Push Topic	72
PushTopic Considerations	72
PushTopic Notification Message Order	73
Considerations for Multiple Notifications in the Same Transaction	74
PushTopic Streaming Allocations	76
Reference: PushTopic	77
<b>Chapter 5: Generic Events (Legacy)</b>	<b>78</b>
Replay Generic Streaming Events with Durable Generic Streaming	79
Generic Streaming Quick Start	79
Create a Streaming Channel	79
Run a Java Client with Username and Password Login	80
Run a Java Client with OAuth Bearer Token Login	81
Generate Events Using REST	82
Generic Streaming Allocations	83
Reference: StreamingChannel	83
Reference: Streaming Channel Push REST API	84
<b>Chapter 6: Monitoring Event Usage</b>	<b>87</b>
Monitor PushTopic Event Usage in the UI	88
Monitor PushTopic and Generic Event Usage with the REST API	88

# CHAPTER 1 Getting Started with Streaming API

## In this chapter ...

- [Push Technology](#)
- [Bayeux Protocol, CometD, and Long Polling](#)
- [Streaming API Terms](#)
- [How the Client Connects](#)
- [Message Reliability](#)
- [Message Durability](#)
- [Streaming Event Features](#)
- [Streaming API vs. Pub/Sub API](#)
- [Differences Between Change Events Received with Streaming API vs. Pub/Sub API](#)
- [API End-of-Life Policy](#)

Streaming API enables streaming of events using push technology and provides a subscription mechanism for receiving events in near real time. The Streaming API subscription mechanism supports multiple types of events, including PushTopic events, generic events, platform events, and change data capture events.

 **Important:** If you're writing an app for publishing and subscribing to platform events and change data capture events, we recommend you use [Pub/Sub API](#) instead of Streaming API. Pub/Sub API is a newer API. Based on gRPC API and HTTP/2, Pub/Sub API efficiently publishes and delivers binary event messages.


Consider these applications for Streaming API.

### Applications That Poll Frequently

Applications that have constant polling action against the Salesforce infrastructure consume unnecessary API calls and processing time. They can benefit from Streaming API because it reduces the number of requests that return no data.

### General Notification

Applications that require general notification of data changes in an org. By using Streaming API, these applications can reduce the number of API calls and improve performance.

 **Note:** You can use Streaming API with any Salesforce org as long as you enable the API. To ensure continuity during instance refreshes and org migrations, we recommend using your org's My Domain login URL instead of its instance in the Streaming API endpoint.

## Push Technology

---

Push technology, also called the publish/subscribe model, transfers information that is initiated from a server to the client. This type of communication is the opposite of pull technology in which a request for information is made from a client to the server.

The information sent by the server is typically specified in advance. When using a PushTopic event, you specify the information that the client receives by creating a PushTopic with specific criteria. The client then subscribes to the PushTopic channel and is notified of events that match the PushTopic criteria. When using a platform event, you first define a platform event and its fields in Salesforce. Then you publish the platform event. The client subscribes to the platform event channel and gets notified of the published event message.

In push technology, the server pushes out information to the client after the client has subscribed to a channel of information. For the client to receive the information, the client must maintain a connection to the server (Salesforce). Streaming API uses the Bayeux protocol and CometD, so the client-to-server connection is maintained through long polling.

## Bayeux Protocol, CometD, and Long Polling

---

Streaming API uses the Bayeux protocol and CometD for long polling.

- Bayeux is a protocol for transporting asynchronous messages, primarily over HTTP.
- CometD is a scalable HTTP-based event routing bus that uses an AJAX push technology pattern known as Comet. It implements the Bayeux protocol.
- Long polling, also called Comet programming, allows emulation of an information push from a server to a client. Similar to a normal poll, the client connects and requests information from the server. However, instead of sending an empty response if information isn't available, the server holds the request and waits until information is available (an event occurs). The server then sends a complete response to the client. The client then immediately re-requests information. The client continually maintains a connection to the server, so it's always waiting to receive a response. In the case of server timeouts, the client connects again and starts over.

If you're not familiar with long polling, Bayeux, or CometD, review the [CometD documentation](#).

Streaming API supports the following CometD methods:

Method	Description
<code>connect</code>	The client connects to the server.
<code>disconnect</code>	The client disconnects from the server.
<code>handshake</code>	The client performs a handshake with the server and establishes a long polling connection.
<code>subscribe</code>	The client subscribes to a channel defined by a PushTopic. After the client subscribes, it can receive messages from that channel. You must successfully call the <code>handshake</code> method before you can subscribe to a channel.
<code>unsubscribe</code>	The client unsubscribes from a channel.

## Streaming API Terms

---

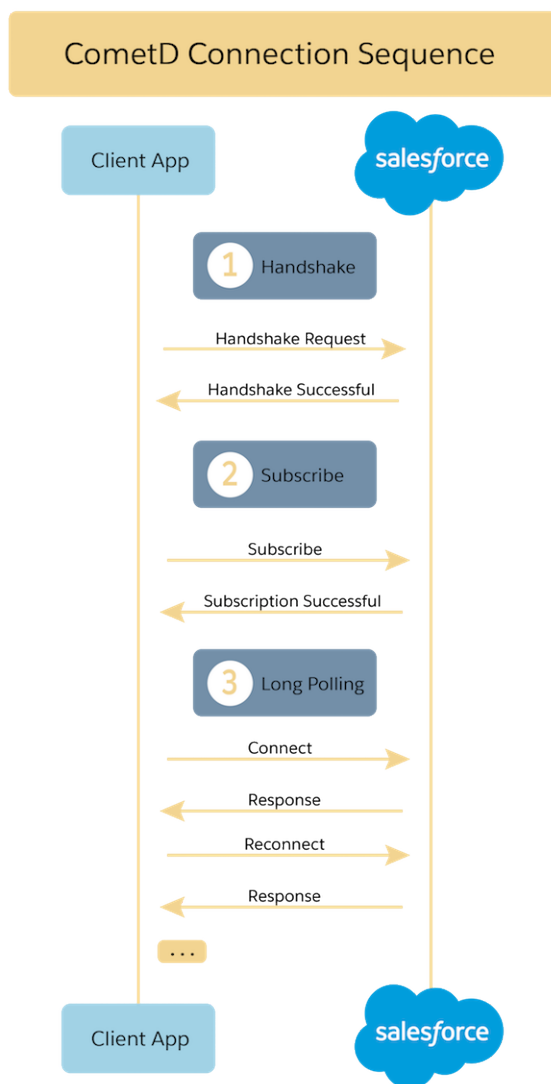
Learn about terms used for Streaming API.

Term	Description
Event	The creation, update, delete, or undelete of a record. Each event might trigger a notification.
Notification	A message in response to an event. The notification is sent to a channel to which one or more clients are subscribed.
PushTopic	A PushTopic triggers notifications for changes in Salesforce records resulting from a create, update, delete, or undelete operation. A PushTopic notification is based on the criteria that you specify in the PushTopic record and the SOQL query that you define. Only the fields specified in the query are included in the notification. The PushTopic defines a subscription channel.
Channel	A stream of events to which a client can subscribe to receive event notifications.
Event Bus	A conduit in which a publisher sends an event notification. Event subscribers subscribe to a channel in the event bus to receive event notifications. The event bus supports replaying stored event messages.
Platform Event	A Salesforce entity that represents the definition of the custom data that you send in a platform event message. You create a platform event and define its fields in Salesforce. The subscription channel is based on the platform event name.
Change Data Capture Event	Similar to a PushTopic, Change Data Capture triggers notifications for changes in Salesforce records resulting from a create, update, delete, or undelete operation. Unlike a PushTopic, Change Data Capture sends all changed fields of a record and doesn't require you to specify the fields in a query. Also, Change Data Capture sends information about the change in headers.

## How the Client Connects

Streaming API uses the HTTP/1.1 request-response model and the Bayeux protocol (CometD implementation). A Bayeux client connects to Streaming API in multiple stages.

1. CometD sends a handshake request.
2. After a successful handshake, your custom listener on the `/meta/handshake` channel sends a subscription request to a channel.
3. CometD maintains the connection by using [long polling](#).



The client receives events from the server while it maintains a long-lived connection. CometD performs the handshake, connection, and reconnection requests. Your custom code performs other operations, such as subscription. The client reconnects for the following conditions.

#### After Receiving Events

If the client receives events, the client must reconnect immediately using CometD to receive the next set of events. If the reconnection doesn't occur within 40 seconds, the server expires the subscription, and the connection closes. The client must start over with a handshake and subscribe again using your custom `/meta/handshake` channel listener.

#### When No Events Are Received

If no events are generated while the client is waiting and the server closes the connection, CometD must reconnect within 110 seconds. The Bayeux server sends a response to the client that contains the reconnect deadline of 110 seconds in the `advice` field. If the client doesn't reconnect within the expected time, the server removes the client's CometD session.

**After a Network Failure**

If a long-lived connection is lost due to unexpected network disruption, CometD attempts to reconnect. If this reconnection is successful, clients must resubscribe, because this new connection has gone through a rehandshake that removes previous subscribers. Clients can listen to the `/meta/handshake` meta channel to receive notifications when a connection is lost and reestablished. For more information, see [Short Network Failures](#) and [Long Network Failures or Server Failures](#) in the [CometD Reference Documentation](#).

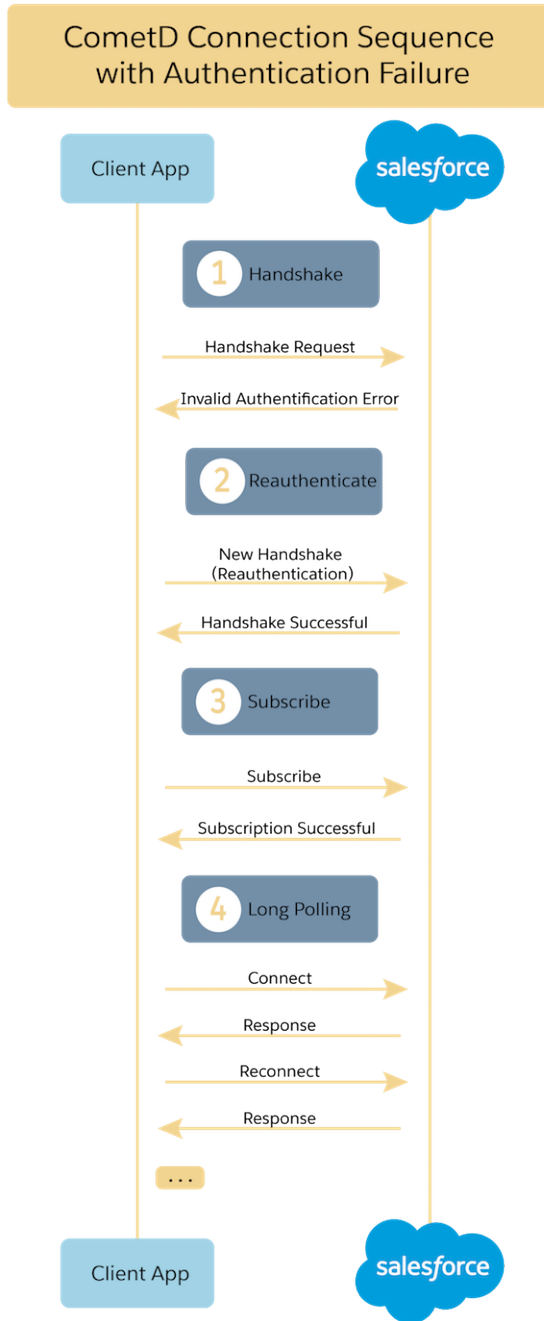
**After Invalid Authentication**

Client authentication can sometimes become invalid, for example, when the OAuth token is revoked or the Salesforce session is invalidated by a Salesforce admin. Streaming API regularly validates the OAuth token or session ID while the client is connected. If client authentication is not valid, the client is notified with the `401::Authentication invalid` error and an `advice` field containing `reconnect=none`. After receiving the error notification in the channel listener, the client must reauthenticate and reconnect to receive new events.




**Note:** Invalidated client authentication doesn't include Salesforce session expiration. The Salesforce session never expires in a CometD client. Salesforce keeps extending the timeout interval as long as the client stays connected.

This diagram shows how a CometD client connects to Salesforce after it encounters an authentication error.



For details about these steps, see [Bayeux Protocol](#), [CometD](#), and [Long Polling](#).

 **Note:** The maximum size of the HTTP request post body that the server can accept from the client is 32,768 bytes, for example, when you call the CometD `subscribe` or `connect` methods. If the request message exceeds this size, the following error is

returned in the response: 413 Maximum Request Size Exceeded. To keep requests within the size limit, avoid sending multiple messages in a single request.

SEE ALSO:

[Handling Streaming API Errors](#)

[Streaming API Error Codes](#)

## Message Reliability

---

For clients subscribed with API version 37.0 or later, Streaming API provides reliable message delivery by enabling you to replay past events through durable streaming. Clients subscribed with API version 36.0 or earlier might not receive all messages in some situations.



**Note:** When using the Streaming API endpoint, note these important considerations.

- Durable streaming is supported when clients subscribe at the Streaming API endpoint using API version 37.0 or later. The PushTopic or platform event version affects only the fields available in the event message, but doesn't affect the client subscription version.
- To ensure continuity during instance refreshes and org migrations, we recommend using your org's My Domain login URL in the Streaming API endpoint.

In API version 37.0 and later, Streaming API stores events for 24 hours, enabling you to replay past events. With durable streaming, messages aren't lost when a client is disconnected or isn't subscribed. When the client subscribes again, it can fetch past events that are within the 24-hour retention period. The ability to replay past events provides reliable message delivery.

In API version 36.0 and earlier, Streaming API doesn't maintain client state nor keeps track of what's delivered. The client might not receive messages for several reasons, including:

- When a client first subscribes or reconnects, it might not receive messages that were processed while it wasn't subscribed to the channel.
- When a client disconnects and starts a new handshake, it could be working with a different application server, so it receives only new messages from that point on.
- Some events are dropped when the system is being heavily used.
- If an application server is stopped, all messages being processed but not yet sent are lost. Clients connected to that application server are disconnected. To receive notifications, the client must reconnect and subscribe to the topic channel.

## Message Durability

---

Salesforce stores PushTopic events, generic events, and standard-volume events for 24 hours and high-volume events for 72 hours. High-volume events include platform events and change data capture events. Standard-volume events are no longer available and include only events defined before Spring '19. With API version 37.0 and later, you can retrieve events that are within the retention window through durable streaming.

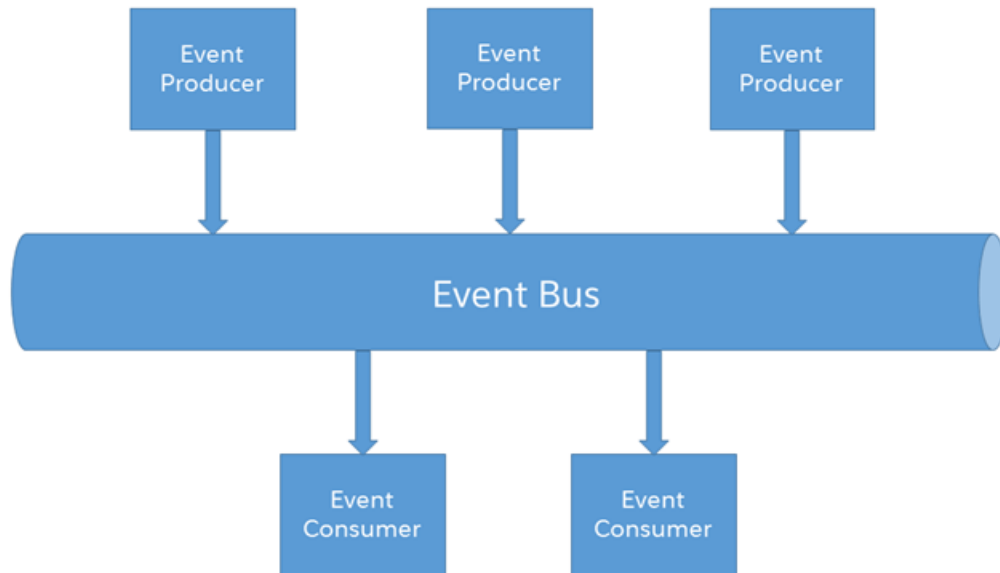


**Tip:** This page describes the event message retention within the context of Streaming API. Pub/Sub API is a newer API for publishing and subscribing to events. To learn about Pub/Sub API, check out the [Pub/Sub API Documentation](#).

After the retention period, events are purged from the event bus. The purging process sometimes starts later, and as a result, high-volume platform events and change data capture events that are older than 72 hours can still be available. Salesforce doesn't guarantee the storage of events beyond the retention period of 72 hours.

## Event Bus

With API version 37.0 and later, events are published to the event bus. Subscribers retrieve events from a channel on the event bus, including past events that are stored temporarily. The event bus decouples event publishers from event subscribers.



## Event Replay Process

Each event message is assigned an opaque ID contained in the `ReplayId` field. The `ReplayId` field value, which is populated by the system when the event is delivered to subscribers, refers to the position of the event in the event stream. Replay ID values aren't guaranteed to be contiguous for consecutive events. A subscriber can store a replay ID value and use it on resubscription to retrieve events that are within the retention window. For example, a subscriber can retrieve missed events after a connection failure. Subscribers must not compute new replay IDs based on a stored replay ID to refer to other events in the stream.

To uniquely identify a platform event message, use the `EventUuid` system field and not the `ReplayId` field. The `ReplayId` field isn't guaranteed to be unique when Salesforce maintenance activities occur, such as an org migration. The `EventUuid` field is always unique.

These examples show the contents of event messages with the `replayId` field. Even though the `replayId` fields in the examples contain numbers, they're opaque fields. We recommend that you don't assume that they always contain numbers. It's best that you store the `replayId` values as bytes.

Platform event messages contain a replay ID when delivered to a CometD client. This JSON message shows the `replayId` field in the event object for the `Low_Ink__e` platform event.

```
{
  "data": {
    "schema": "dffQ2QLzDNHqwB8_sHMxdA",
    "payload": {
```

```

    "CreateDate": "2023-04-09T18:31:40.517Z",
    "CreatedById": "005D0000001cSZs",
    "Printer_Model__c": "XZO-5",
    "Serial_Number__c": "12345",
    "Ink_Percentage__c": 0.2
  },
  "event": {
    "EventUuid": "2ec0e371-1395-457f-9275-be1b527a72f7",
    "replayId": 2112
  }
},
"channel": "/event/Low_Ink__e"
}

```

This JSON message shows the `replayId` field in the event object for a Change Data Capture event.

```

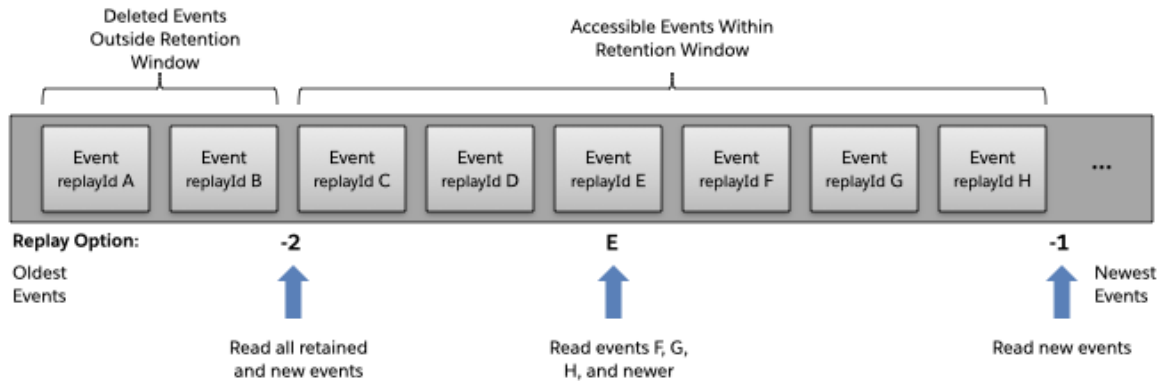
{
  "data": {
    "schema": "IeRuaY6cbI_HsV8Rv1Mc5g",
    "payload": {
      "ChangeEventHeader": {
        "entityName": "Account",
        "recordIds": [
          "<record_ID>"
        ],
        "changeType": "CREATE",
        "changeOrigin": "com.salesforce.core",
        "transactionKey": "001b7375-0086-250e-e6ca-b99bc3a8b69f",
        "sequenceNumber": 1,
        "isTransactionEnd": true,
        "commitTimestamp": 1501010206653,
        "commitNumber": 92847272780,
        "commitUser": "<User_ID>"
      },
      "Name": "Acme",
      "Description": "Everyone is talking about the cloud. But what does it mean?",
      "OwnerId": "<Owner_ID>",
      "CreateDate": "2017-07-25T19:16:44Z",
      "CreatedById": "<User_ID>",
      "LastModifiedDate": "2017-07-25T19:16:44Z",
      "LastModifiedById": "<User_ID>"
    },
    "event": {
      "replayId": 6421
    }
  },
  "channel": "/data/ChangeEvents"
}

```


## Replaying Events

A subscriber can choose which events to receive, such as all events within the retention window or starting after a particular event. The default is to receive only the new events sent after subscribing. Events outside the retention period are discarded.

This high-level diagram shows how event consumers can read a stream of events by using various replay options.



**Table 1: Replay Options**

Replay Option	Description	Usage
Replay ID	Subscriber receives all stored events after the event specified by its <code>replayId</code> value and new events.	Catch up on missed events after a certain event message, for example, after a connection failure. To subscribe with a specific replay ID, save the replay ID of the event message after which you want to retrieve stored events. Then use this replay ID when you resubscribe.
-1	(Default if no replay option is specified.) Subscriber receives new events that are broadcast after the client subscribes.	We recommend that you subscribe with the -1 option to receive new event messages. To get earlier event messages, we recommend that you use a specific replay ID that you saved earlier.
-2	Subscriber receives all events, including past events that are within the retention window and new events.	Catch up on missed events and retrieve all stored events, for example, after a connection failure.  <div>  <b>Important:</b> Use this option sparingly. Subscribing with the -2 option when a large number of event messages are stored can slow performance. </div>

To replay events, use the Streaming API endpoint.

<https://MyDomainName.my.salesforce.com/cometd/63.0/>



**Note:** When using the Streaming API endpoint, note these important considerations.

- Durable streaming is supported when clients subscribe at the Streaming API endpoint using API version 37.0 or later. The PushTopic or platform event version affects only the fields available in the event message, but doesn't affect the client subscription version.
- To ensure continuity during instance refreshes and org migrations, we recommend using your org's My Domain login URL in the Streaming API endpoint.

The replay mechanism is implemented in a Salesforce-provided CometD extension. A sample extension is provided in JavaScript and another in Java. For example, you can register the extension as follows in JavaScript.

```
// Register streaming extension
var replayExtension = new cometdReplayExtension();
replayExtension.setChannel(<Streaming Channel to Subscribe to>);
replayExtension.setReplay(<Event Replay Option>);
cometd.registerExtension('myReplayExtensionName', replayExtension);
```

 **Note:**

- The argument passed to `setReplay()` is one of the replay options. We recommend that clients subscribe with the `-1` option to receive new events or with a specific replay ID. If the channel contains many event messages, subscribing frequently with the `-2` option can cause performance issues.
- The first argument passed to `registerExtension()` is the name of the replay extension in your code. In the example, it's set to `myExtensionName`, but it can be any string. You use this name to unregister the extension later on.
- If the `setReplay()` function isn't called, or the CometD extension isn't registered, only new events are sent to the subscriber, which is the same as the `-1` option.

After calling the `setReplay()` function on the extension, the events that the subscriber receives depend on the replay value parameter passed to `setReplay()`.

After a client times out because it hasn't reconnected within 40 seconds or a network failure has occurred, it attempts a new handshake request and reconnects. The replay extension saves the replay ID of the last message received and uses it when resubscribing. That way, the client receives only messages that were sent after the timeout and doesn't receive duplicate messages that were sent earlier.

## Code Samples

### Java Sample

For a Java client sample that uses the CometD extension, see [Example: Subscribe to and Replay Events Using a Java Client \(EMP Connector\)](#).

### Lightning Component Sample

For a sample that uses the `empApi` Lightning component, see [Example: Subscribe to and Replay Events Using a Lightning Component](#).

### Visualforce Sample

For a sample and code walkthrough that uses Visualforce and a CometD extension in JavaScript, see [Example: Subscribe to and Replay Events Using a Visualforce Page](#).

### SEE ALSO:

[Bayeux Protocol, CometD, and Long Polling](#)

[Clients and Timeouts](#)

[Platform Events Developer Guide: Platform Event Fields](#)

## Streaming Event Features

The Lightning Platform offers several types of streaming events. To determine which event meets your use case, compare the features of the various events.

**Important:** PushTopic and generic events are legacy products. Salesforce no longer enhances them with new features and provides limited support for them. Instead of PushTopic events, consider using Change Data Capture events. Instead of generic events, consider using Platform Events. To learn about Change Data Capture events, see the [Change Data Capture Developer Guide](#) and the [Change Data Capture Basics](#) Trailhead module. To learn about Platform Events, see [Platform Events Developer Guide](#) and the [Platform Events Basics Trailhead module](#).

Feature	Change Data Capture Event	Platform Event	PushTopic Event (Legacy)	Generic Event (Legacy)
Define a custom schema as strongly typed fields	N/A	✓	N/A	✗
Include user-defined payloads	N/A	✓	N/A	✓
Publish custom events via one or more APIs	N/A	✓	N/A	✓
Publish events via Apex	N/A	✓	N/A	✗
Publish declaratively using Process Builder and Flow Builder	N/A	✓	N/A	✗
Publish to specific users	N/A	✗	N/A	✓
Subscribe via CometD using JavaScript, Java, and other languages	✓	✓	✓	✓
Subscribe via Pub/Sub API	✓	✓	✗	✗
Subscribe via Apex triggers	✓	✓	✗	✗
Filter subscriptions	✓	✓	✓	✗
Receive auto-published event notifications for Salesforce record changes	✓	N/A	✓	N/A
Choose the fields to include in event notifications for Salesforce record changes	✗	N/A	✓	N/A
Receive a versioned event schema	✓	✓	✗	✗
Get field-level security	✓	✗	✓	✗
Get record-sharing support	✗	N/A	✓	N/A
Encrypt field data with Shield Platform Encryption	✓	✓	✗	✗
Replay retained event notifications	✓	✓	✓	✓
Event retention period	3 days	3 days*	1 day	1 day

\* Standard-volume platform events are retained for 1 day.

SEE ALSO:

[Platform Events Developer Guide](#)

[Change Data Capture Developer Guide](#)

## Streaming API vs. Pub/Sub API

Pub/Sub API is a newer API that you can use to publish and subscribe to platform events and change data capture events. Based on gRPC API and HTTP/2, Pub/Sub API efficiently publishes and delivers binary event messages, and supports multiple programming languages. Before Pub/Sub API was introduced, the only way to subscribe to events in an external client was with Streaming API. This table compares some features of the two subscription APIs.

	Streaming API	Pub/Sub API
Event encoding	JSON	Binary ( <a href="#">Apache Avro</a> )
Protocol	<a href="#">CometD</a> , an implementation of the Bayeux protocol.	<a href="#">gRPC</a> and protocol buffers
Subscription model	Push-based—The server delivers events as they become available in the event bus.	Pull-based—The client requests the number of events to receive as they become available in the event bus.
Subscription flow control	The client doesn't control the number of events delivered from the server.	The client specifies how many events to receive for a Subscribe call based on event processing speed.
Publish status	Not supported with Streaming API. When using data APIs to publish events, the event queueing result is returned. When system resources are available, queued events are published. The final result isn't returned to the client.	The final publishing result returned indicates that the event was successfully published or that the publish failed.

SEE ALSO:

[Pub/Sub API Developer Guide](#)

## Differences Between Change Events Received with Streaming API vs. Pub/Sub API

The format of change events received with Streaming API differs from change events received with Pub/Sub API. Streaming API delivers the entire event message in JSON format while Pub/Sub API delivers the event payload in the Apache Avro binary format.

**Important:** If you're writing an app for publishing and subscribing to platform events and change data capture events, we recommend you use [Pub/Sub API](#) instead of Streaming API. Pub/Sub API is a newer API. Based on gRPC API and HTTP/2, Pub/Sub API efficiently publishes and delivers binary event messages.

Event Message Content	Streaming API	Pub/Sub API
Event message format	The entire event message is in JSON format.	The event payload is in the Apache Avro binary format.  The client can retrieve the schema, replay ID, and payload from the received event separately and decode the payload to obtain

Event Message Content	Streaming API	Pub/Sub API
		the <code>ChangeEventHeader</code> and record fields. For more information, see <a href="#">Event Data Serialization with Apache Avro</a> in the Pub/Sub API documentation.
Record fields	Null fields and unchanged fields are excluded from the event message.	All the record fields, including the unchanged fields. Unchanged fields are null even if they have a value in the Salesforce record. Unpopulated fields are also null.
<code>ChangeEventHeader</code> fields	All the fields in <a href="#">ChangeEventHeader Fields</a> except for <code>changedFields</code> and <code>nulledFields</code> .	In addition to the fields that are sent in Streaming API, Pub/Sub API includes these fields: <ul style="list-style-type: none"> <li><code>changedFields</code></li> <li><code>nulledFields</code></li> </ul> See <a href="#">ChangeEventHeader Fields</a> in the <i>Change Data Capture Developer Guide</i> .  Also, <code>nulledFields</code> , <code>diffFields</code> , and <code>changedFields</code> require further processing. For more information, see <a href="#">Event Deserialization Considerations</a> in the Pub/Sub API documentation.
Date field format	Date fields contain the date in UTC format.	Date fields are in Epoch time. They can be converted to another date format for readability.

## Change Event Example in Streaming API (CometD)

This event message is sent for a new account in a CometD client.


```
{
  "schema": "V2MGwDA2aIP0yQS98S2L6Q",
  "payload": {
    "LastModifiedDate": "2024-04-09T20:26:27.000Z",
    "Description": "Sample account record.",
    "CleanStatus": "Pending",
    "OwnerId": "0055f000005mc66AAA",
    "CreatedById": "0055f000005mc66AAA",
    "ChangeEventHeader": {
      "commitNumber": 1082990428293,
      "commitUser": "0055f000005mc66AAA",
      "sequenceNumber": 1,
      "entityName": "Account",
      "changeType": "CREATE",
      "changedFields": [],
      "changeOrigin": "com/salesforce/api/soap/60.0;client=SfdcInternalAPI/",
      "transactionKey": "0001ae4b-7024-a64a-87f0-f2c8cebdca76",

```

```

    "commitTimestamp": 1712694387000,
    "recordIds": [
      "0015f00002J9YaUAAV"
    ]
  },
  "CreatedDate": "2024-04-09T20:26:27.000Z",
  "LastModifiedById": "0055f000005mc66AAA",
  "Name": "Acme"
},
"event": {
  "replayId": 37904280
}
}

```

 **Note:** The order of the fields in the JSON event message received in a Streaming (CometD) isn't guaranteed. The order is based on the underlying Apache Avro schema that change events are based on. When an event is expanded into JSON format, the order of the fields doesn't always match the schema depending on the client used to receive the event.

## Change Event Example in Pub/Sub API

This event message is sent for a new account in a Pub/Sub API client.

```

{
  "ChangeEventHeader": {
    "entityName": "Account",
    "recordIds": [
      "0015f00002J9YYEAA3"
    ],
    "changeType": "CREATE",
    "changeOrigin": "com/salesforce/api/soap/60.0;client=SfdcInternalAPI/",
    "transactionKey": "0001ade9-3f74-0b99-dbc4-42e73424b774",
    "sequenceNumber": 1,
    "commitTimestamp": 1712693965000,
    "commitNumber": 1082985383811,
    "commitUser": "0055f000005mc66AAA",
    "nulledFields": [],
    "diffFields": [],
    "changedFields": []
  },
  "Name": "Acme",
  "Type": null,
  "ParentId": null,
  "BillingAddress": null,
  "ShippingAddress": null,
  "Phone": null,
  "Fax": null,
  "AccountNumber": null,
  "Website": null,
  "Sic": null,
  "Industry": null,
  "AnnualRevenue": null,
  "NumberOfEmployees": null,
  "Ownership": null,
  "TickerSymbol": null,

```

```

    "Description": "Sample account record.",
    "Rating": null,
    "Site": null,
    "OwnerId": "0055f000005mc66AAA",
    "CreatedDate": 1712693965000,
    "CreatedById": "0055f000005mc66AAA",
    "LastModifiedDate": 1712693965000,
    "LastModifiedById": "0055f000005mc66AAA",
    "Jigsaw": null,
    "JigsawCompanyId": null,
    "CleanStatus": "Pending",
    "AccountSource": null,
    "DunsNumber": null,
    "Tradestyle": null,
    "NaicsCode": null,
    "NaicsDesc": null,
    "YearStarted": null,
    "SicDesc": null,
    "DandbCompanyId": null,
    "OperatingHoursId": null,
    "CustomerPriority__c": null,
    "SLA__c": null,
    "Active__c": null,
    "NumberOfLocations__c": null,
    "UpsellOpportunity__c": null,
    "SLASerialNumber__c": null,
    "SLAExpirationDate__c": null
  }

```

SEE ALSO:

[Change Data Capture Developer Guide](#)

## API End-of-Life Policy

Check out the policy for retiring API versions, and which Streaming API versions are supported, deprecated, or retired.

Salesforce is committed to supporting each API version for a minimum of 3 years from the date of first release. To improve the quality and performance of the API, versions that are over 3 years old sometimes are no longer supported.

Salesforce notifies customers who use an API version scheduled for deprecation at least 1 year before support for the version ends.

Streaming API Versions	Version Support Status	Version Retirement Info
Versions 37.0 through 63.0	Supported.	
Versions 23.0 through 36.0	<p>Streaming API was introduced in API version 23.0. As of Winter '25, versions 23.0 through 36.0 are no longer available.</p> <p>If a client connects using a retired API version, the system routes the request to the latest API version.</p>	<a href="#">Streaming API Versions 23.0 through 36.0 Retirement</a>

## CHAPTER 2 Code Examples

### In this chapter ...


- Example: Subscribe to and Replay Events Using a Java Client (EMP Connector)
- Example: Subscribe to and Replay Events Using a Lightning Component
- Example: Subscribe to and Replay Events Using a Visualforce Page
- Example: Interactive Visualforce Page without Replay
- Example: Authentication

Check out code examples for streaming events in Java, Aura components, and Visualforce.

## Example: Subscribe to and Replay Events Using a Java Client (EMP Connector)

---

The Java sample uses a library called Enterprise Messaging Platform (EMP) Connector. EMP Connector is a thin wrapper around the CometD library. It hides the complexity of creating a CometD client and subscribing to Streaming API in Java. The example subscribes to a channel, receives notifications, and supports replaying events with durable streaming.


 **Important:** The EMP Connector sample is deprecated and will be archived in the future. The EMP Connector code is an example only and isn't intended for production environments. It hasn't been rigorously tested nor performance tested for throughput and scale.

Are you looking for a code sample for subscribing to platform events and change events? Check out the [Java Quick Start for Pub/Sub API](#) in the *Pub/Sub API Guide*.

The CometD-based subscription mechanism in EMP Connector can receive any type of Salesforce event. Just pass in the channel name of the desired event. For example, the events that EMP Connector can receive include:

- Platform events
- Change Data Capture events
- PushTopic events (legacy)
- Generic events (legacy)

EMP Connector is based on Java and uses CometD version 3.1.0. It supports username and password authentication and OAuth bearer token authentication.

 **Note:** The example requires API version 37.0 or later. For a code example that supports earlier API versions, refer to an earlier version of this documentation.

### IN THIS SECTION:

#### [Prerequisites](#)

You need access and appropriate permissions to complete the code example.

#### [Step 1: Create an Object](#)

Create an Invoice Statement object from the user interface.

#### [Step 2: Create a PushTopic \(Legacy\)](#)

Create a PushTopic in the Developer Console. Event notifications are generated for updates that match the query.

#### [Step 3: Download and Build the Project](#)

Before you can run the connector examples, download the Java source files and build the Java project.

#### [Step 4: Use the Connector with Username and Password Login](#)

Now that you've downloaded and built EMP Connector, use it to connect to CometD and subscribe to the PushTopic.

#### [\(Optional\) Step 5: Use the Connector with OAuth Bearer Token Login](#)

You can use the connector with OAuth authentication as an alternative to username and password authentication. This step is optional and requires an OAuth token.

[Learn More About EMP Connector](#)

Let's take a closer look at the components of EMP Connector.

SEE ALSO:

[Platform Events Developer Guide](#)

[Change Data Capture Developer Guide](#)

## Prerequisites

You need access and appropriate permissions to complete the code example.

- Java Development Kit 8 or later (see [Java Downloads](#))
- Eclipse IDE for Java Developers (get a recent version from <http://www.eclipse.org/downloads/eclipse-packages/>). This example walks you through the steps of building the project with the Eclipse IDE but you can use your preferred IDE to build the Java client.
- To run the tool from the command line: Apache Maven (see <https://maven.apache.org/index.html>)
- Access to a Developer Edition org

To create a Developer Edition org, go to [developer.salesforce.com/signup](https://developer.salesforce.com/signup) and follow the instructions for signing up for a Developer Edition organization.

- The API Enabled permission must be enabled for your Developer Edition org. This permission is enabled by default, but an admin might have changed it.
- The Streaming API permission must be enabled in Setup, in the User Interface page. This permission is enabled by default, but an admin might have been changed it.
- The logged-in user must have Read permission on the PushTopic standard object to receive notifications.
- The logged-in user must have Create permission on the PushTopic standard object to create and manage PushTopic records.
- The logged-in user must have Author Apex permission to create a PushTopic from the Developer Console.

## Step 1: Create an Object

Create an Invoice Statement object from the user interface.

1. In Setup, from your management settings for custom objects, if you're using Salesforce Classic, click **New Custom Object**, or if you're using Lightning Experience, select **Create > Custom Object**.
2. Define the custom object.
  - In the Label field, enter *Invoice Statement*.
  - In the Plural Label field, enter *Invoice Statements*.
  - Select **Starts with vowel sound**.
  - In the Record Name field, enter *Invoice Number*.
  - In the Data Type field, select **Auto Number**.
  - In the Display Format field, enter *INV- { 0000 }*.
  - In the Starting Number field, enter *1*.
3. Save your work.
4. Add a Status field.

- a. In Custom Fields & Relationships, click **New**.
- b. For Data Type, select `Picklist`, and then click **Next**.
- c. In the Field Label field, enter `Status`.
- d. For Values, select **Enter values, with each value separated by a new line**.
- e. Enter these picklist values with each entry on its own line.


```
Open
Closed
Negotiating
Pending
```

- f. Click **Use first value as default value**.
  - g. Click **Next**.
  - h. For field-level security, select `Read Only`, and then click **Next**.
  - i. Click **Save & New** to save this field and create another.
5. Now create an optional Description field.
    - a. In the Data Type field, select `Text Area`, and then click **Next**.
    - b. In the Field Label and Field Name fields, enter `Description`.
    - c. Click **Next**, accept the defaults, and then click **Next** again.
    - d. Click **Save** to go the detail page for the Invoice Statement object.

Your Invoice Statement object now shows two custom fields.


## Step 2: Create a PushTopic (Legacy)

Create a PushTopic in the Developer Console. Event notifications are generated for updates that match the query.


 **Important:** PushTopic events is a legacy product. Salesforce no longer enhances PushTopic events with new features and provides limited support for this product. Instead of PushTopic events, consider using Change Data Capture events. To subscribe to change events, see [Java Quick Start for Pub/Sub API](#) in the *Pub/Sub API Guide*. To learn about Change Data Capture, see the [Change Data Capture Developer Guide](#) and the [Change Data Capture Basics](#) Trailhead module.

1. Open the Developer Console.
2. Click **Debug > Open Execute Anonymous Window**.
3. In the Enter Apex Code window, paste in the following Apex code, and click **Execute**.

```
PushTopic pushTopic = new PushTopic();
pushTopic.Name = 'InvoiceStatementUpdates';
pushTopic.Query = 'SELECT Id, Name, Status__c, Description__c FROM Invoice_Statement__c';
pushTopic.ApiVersion = 63.0;
pushTopic.NotifyForOperationCreate = true;
pushTopic.NotifyForOperationUpdate = true;
pushTopic.NotifyForOperationUndelete = true;
pushTopic.NotifyForOperationDelete = true;
pushTopic.NotifyForFields = 'Referenced';
insert pushTopic;
```

 **Note:** If your organization has a namespace prefix defined, then you'll need to preface the custom object and field names with that namespace when you define the PushTopic query. For example, `SELECT Id, Name, namespace__Status__c, namespace__Description__c FROM namespace__Invoice_Statement__c`.

Because `NotifyForOperationCreate`, `NotifyForOperationUpdate`, `NotifyForOperationDelete` and `NotifyForOperationUndelete` are set to `true`, Streaming API evaluates records that are created, updated, deleted, or undeleted and generates a notification if the record matches the PushTopic query. Because `NotifyForFields` is set to `Referenced`, Streaming API will use fields in both the `SELECT` clause and the `WHERE` clause to generate a notification. Whenever the fields `Name`, `Status__c`, or `Description__c` are updated, a notification will be generated on this channel. For more information about `NotifyForOperationCreate`, `NotifyForOperationUpdate`, `NotifyForOperationDelete`, `NotifyForOperationUndelete`, and `NotifyForFields`, see [Event Notification Rules](#).

 **Note:** In API version 28.0 and earlier, notifications are only generated when records are created or updated. The `NotifyForOperationCreate`, `NotifyForOperationUpdate`, `NotifyForOperationDelete`, and `NotifyForOperationUndelete` fields are unavailable and the `NotifyForOperations` enum field is used instead to set which record events generate a notification. For more information see [PushTopic](#).

## Step 3: Download and Build the Project


Before you can run the connector examples, download the Java source files and build the Java project.

### Prerequisites:

- Java Development Kit 8 or later (see [Java Downloads](#))
- Eclipse IDE for Java Developers (get a recent version from <http://www.eclipse.org/downloads/eclipse-packages/>). This example walks you through the steps of building the project with the Eclipse IDE but you can use your preferred IDE to build the Java client.

The EMP Connector project includes examples in the [GitHub repository's example folder](#) that use the connector to log in and subscribe to events. In the next steps, you run the following examples locally on your system.

- [LoginExample.java](#)
- [BearerTokenExample.java](#)

 **Note:** `LoginExample.java` logs in to your Developer Edition org instance using the instanced login URL such as `https://na139.salesforce.com`. To pass in a different login URL, such as your org's My Domain login URL for production or a sandbox, use `DevLoginExample.java` instead. We recommend using `DevLoginExample.java` because My Domain login URLs provide an extra layer of security. `DevLoginExample.java` also provides debug logging for the Bayeux messages received.

To download and build the EMP connector project:

1. To download the project files, do one of the following.
  - Clone the EMP-Connector project using git.

```
git clone https://github.com/forcedotcom/EMP-Connector
```

- Download the project zip file from GitHub, and then extract the zip to a local folder.
2. In Eclipse, import the Maven project from the folder where you cloned or extracted the project.  
The dependencies that are specified in the Maven's `pom.xml` file, such as CometD, are added in the Java project in Eclipse.
  3. If the Java project wasn't automatically built, build it.

If you prefer to run the tool from the command line, generate the JAR file using the Maven command `mvn clean package`. The generated JAR file includes the connector and the example class functionality. The JAR file is a shaded JAR—it contains all dependencies for the connector, so you don't have to download them separately. The JAR file has a `-phat` Maven classifier. You can run the login example from the command line. To run the tool against a Developer Edition org instance without specifying a login URL, use this command, which uses the `LoginExample.java` class by default.

```
$ java -jar target/emp-connector-0.0.1-SNAPSHOT-phat.jar <username> <password> <channel>
[optional_replay_id]
```

To specify a different login URL, such as your org's My Domain login URL, use this command, which references the `DevLoginExample.java` class.

```
$ java -classpath target/emp-connector-0.0.1-SNAPSHOT-phat.jar
com.salesforce.emp.connector.example.DevLoginExample <login_URL> <username> <password>
<channel> [optional_replayId]
```

For `<login_URL>`, use your org's My Domain login URL, including the `https://` prefix. For example, `https://MyDomainName.my.salesforce.com`.

### Open Source Project

EMP Connector is an open-source project, so you can contribute to it with your own enhancements by submitting pull requests to the repository.

## Step 4: Use the Connector with Username and Password Login

Now that you've downloaded and built EMP Connector, use it to connect to CometD and subscribe to the `PushTopic`.

Let's run an example that uses username and password login.

1. In the `/src/main/java/com/salesforce/emp/connector/example` folder, open the `LoginExample.java` source file.
2. Run the `LoginExample` class and provide arguments.
  - a. In Package Explorer, navigate to the `LoginExample.java` file. Right-click the file, and select **Run As > Run Configurations**.
  - b. On the Arguments tab, add values for the following arguments, separated by a space.

Argument	Value
<code>username</code>	Username of the logged-in user
<code>password</code>	Password for the <code>username</code> (or logged-in user)
<code>channel</code>	The channel name for the <code>PushTopic</code> : <code>/topic/InvoiceStatementUpdates</code>



**Note:** This quick start is based on a `PushTopic` event. Alternatively, you can use EMP Connector to listen to any event type. The following lists channel name formats for a sample of streaming events available in the Lightning Platform.

#### Platform event

For a custom platform event—`/event/EventName__e`

For a standard platform event—`/event/EventName`

For a custom channel—`/event/ChannelName__chn`

Argument	Value
	<b>Change Data Capture event</b> For all change events—/data/ChangeEvent For a specific standard object—/data/ <i>ObjectName</i> ChangeEvent For a specific custom object—/data/ <i>CustomObjectName</i> __ChangeEvent For a custom channel—/data/ <i>ChannelName</i> __chn
	<b>PushTopic event</b> /topic/ <i>PushTopicName</i>
	<b>Generic event</b> /u/notifications/ <i>GenericStreamingChannel</i>

c. Click **Run**.

The sample is now subscribed to the event channel and is listening to event notifications. As soon as an event notification is published and received, the tool prints it to the console.

Optionally, to receive different events, you can include a replay ID as the last argument. Valid values are:

- -1 — Get all new events sent after subscription. This option is the default.
- -2 — Get all new events sent after subscription and all past events within the retention window. Use -2 sparingly. If a large volume of event messages is stored, retrieving all event messages can slow performance.
- Specific number — Get all events that occurred after the event with the specified replay ID.

3. In a browser window, create or modify an invoice statement. After you create or change data that corresponds to the query in your PushTopic, the output looks similar to the following.

```
Subscribed: Subscription [/topic/InvoiceStatementUpdates:-1]
Received:
{event={createdDate=2016-12-12T22:31:48.035Z, replayId=1, type=created},
subject={Status__c=Open, Id=a070P00000pn0hyQAA, Name=INV-0001, Description__c=blah}}
Received:
{event={createdDate=2016-12-12T22:32:06.440Z, replayId=2, type=updated},
subject={Status__c=Negotiating, Id=a070P00000pn0hyQAA, Name=INV-0001,
Description__c=blah}}
Received:
{event={createdDate=2016-12-12T22:32:57.404Z, replayId=3, type=created},
subject={Status__c=Open, Id=a070P00000pn0lfQAA, Name=INV-0002, Description__c=Laptops
and accessories.}}
```



**Note:** If you're listening to another event type, the output would look a bit different. Some events require that you publish the notification instead of Salesforce, such as with platform events.

Generally, don't handle usernames and passwords of others when running code in production. In a production environment, delegate the login to OAuth. The next step connects to Streaming API with OAuth.

## (Optional) Step 5: Use the Connector with OAuth Bearer Token Login

You can use the connector with OAuth authentication as an alternative to username and password authentication. This step is optional and requires an OAuth token.

Obtain an OAuth bearer access token for your Salesforce user. You supply this access token in the connector example.

See [Set Up Authorization with OAuth 2.0](#). Also see [Authorize Apps with OAuth](#).

Let's run an example that uses OAuth bearer token login.

1. In the `/src/main/java/com/salesforce/emp/connector/example` folder, open the `BearerTokenExample.java` Java source file.
2. Run the `BearerTokenExample` class, and provide the following argument values.

Argument	Value
<code>url</code>	URL of the Salesforce instance of the logged-in user
<code>token</code>	The access token returned by the OAuth authentication flow
<code>channel</code>	The channel name for the PushTopic: <code>/topic/InvoiceStatementUpdates</code>



**Note:** This quick start is based on a PushTopic event. Alternatively, you can use EMP Connector to listen to any event type. The following lists channel name formats for a sample of streaming events available in the Lightning Platform.

### Platform event

For a custom platform event—`/event/EventName__e`

For a standard platform event—`/event/EventName`

For a custom channel—`/event/ChannelName__chn`

### Change Data Capture event

For all change events—`/data/ChangeEvent`

For a specific standard object—`/data/ObjectNameChangeEvent`

For a specific custom object—`/data/CustomObjectName__ChangeEvent`

For a custom channel—`/data/ChannelName__chn`

### PushTopic event

`/topic/PushTopicName`

### Generic event

`/u/notifications/GenericStreamingChannel`

Optionally, to receive different events, you can include a replay ID as the last argument. Valid values are:

- `-1` — Get all new events sent after subscription. This option is the default.
- `-2` — Get all new events sent after subscription and all past events within the retention window. Use `-2` sparingly. If a large volume of event messages is stored, retrieving all event messages can slow performance.
- Specific number — Get all events that occurred after the event with the specified replay ID.

3. In a browser window, create or modify an invoice statement. After you create or change data that corresponds to the query in your PushTopic, the output looks similar to the following.

```
Subscribed: Subscription [/topic/InvoiceStatementUpdates:-1]
Received:
{event={createdDate=2016-12-12T22:31:48.035Z, replayId=1, type=created},
subject={Status__c=Open, Id=a070P00000pn0hyQAA, Name=INV-0001, Description__c=blah}}
Received:
{event={createdDate=2016-12-12T22:32:06.440Z, replayId=2, type=updated},
subject={Status__c=Negotiating, Id=a070P00000pn0hyQAA, Name=INV-0001,
Description__c=blah}}
Received:
{event={createdDate=2016-12-12T22:32:57.404Z, replayId=3, type=created},
subject={Status__c=Open, Id=a070P00000pn0lfQAA, Name=INV-0002, Description__c=Laptops
and accessories.}}
```



**Note:** If you're listening to another event type, the output would look a bit different. Some events require that you publish the notification instead of Salesforce, such as with platform events.

## Learn More About EMP Connector

Let's take a closer look at the components of EMP Connector.

### Authenticating

The `LoginExample` class logs in to production by default using the passed-in user-credential information.

After initial authentication, `LoginExample` reauthenticates the user if the authentication becomes invalid, such as when a Salesforce session is invalidated or an access token is revoked. `LoginExample` listens to `401::Authentication invalid` error messages that Streaming API sends when the authentication is no longer valid. The class reauthenticates after a 401 error is received. The token provider performs the reauthentication and is set using the `EmpConnector.setBearerTokenProvider()` method.

```
BearerTokenProvider tokenProvider = new BearerTokenProvider(() -> {
    try {
        return login(argv[0], argv[1]);
    } catch (Exception e) {
        e.printStackTrace(System.err);
        System.exit(1);
        throw new RuntimeException(e);
    }
});

BayeuxParameters params = tokenProvider.login();
// . . .
connector.setBearerTokenProvider(tokenProvider);
```

For OAuth authentication, the `BearerTokenExample` uses the `BayeuxParameters` constructor to override the methods in the `BayeuxParameters` class and provides the token and URL values.

```
BayeuxParameters params = new BayeuxParameters() {


    @Override
    public String bearerToken() {
        return "<token>";
    }
};
```

```

    }

    @Override
    public URL host() {
        try {
            return new URL("<URL>");
        } catch (MalformedURLException e) {
            throw new IllegalArgumentException(
                String.format("Unable to create url: %s", argv[0]), e);
        }
    }
};

```

 **Note:** BearerTokenExample doesn't support reauthentication, but you can add this support. Reauthentication is implemented only in LoginExample and DevLoginExample.

## Listening to Events

To listen to events, the connector uses the Java event in a lambda expression. This statement prints the event message to the output for each received event notification. Place this statement before the statement that subscribes to the channel.

```

Consumer<Map<String, Object>> consumer = event -> System.out.println(
    String.format("Received:\n%s", JSON.toString(event)));

```

## Subscribing to a Channel

The EmpConnector class is the main class that exposes the functionality of starting a connection and subscribing. The class contains functions to create a connection, subscribe to a channel, cancel a subscription, and stop a connection.

```

// Instantiate the EMP connector
EmpConnector connector = new EmpConnector(params);

connector.setBearerTokenProvider(tokenProvider);

// Wait for handshake with Streaming API
connector.start().get(5, TimeUnit.SECONDS);

// Subscribe to a channel
// Block and wait for the subscription to succeed for 5 seconds
TopicSubscription subscription = connector.subscribe("<Channel_Name>",
    replayFrom, consumer).get(5, TimeUnit.SECONDS);

```

To end a subscription, call these functions.

```

// Cancel a subscription
subscription.cancel();

// Stop the connector
connector.stop();

```

## Debug Logging

To aid in debugging, the `LoggingListener` class logs Bayeux messages to the console. `BearerTokenExample` and `DevLoginExample` use logging but not `LoginExample`. `DevLoginExample` is part of the EMP Connector GitHub project, but is not covered in this walkthrough. For more information, see the [EMP Connector Readme page](#).

## Example: Subscribe to and Replay Events Using a Lightning Component


Subscribe to event streaming channels with the `empApi` component in your Lightning web component or Aura component. The `empApi` component provides access to methods for subscribing to a streaming channel and listening to event messages.

All types of streaming events are supported, including:

- Platform events
- Change Data Capture events
- PushTopic events (legacy)
- Generic events (legacy)

The `empApi` component uses a shared CometD-based Streaming API connection, enabling you to run multiple streaming apps in the browser for one user. The connection isn't shared across user sessions.

The concurrent CometD client limit applies to the `empApi` component. Each logged-in user using `empApi` counts as one concurrent client. The `empApi` component isn't recommended for apps or sites that are used by a large number of users, such as Experience Cloud sites, because the limit can be reached. This limit is shared with other CometD clients. For more information, see [Platform Event Allocations](#) in the *Platform Events Developer Guide*.

 **Note:** As of Spring '19 (API version 45.0), you can build Lightning components using two programming models: the Lightning Web Components model, and the original Aura Components model. Lightning web components are custom HTML elements built using HTML and modern JavaScript. Lightning web components and Aura components can coexist and interoperate on a page.

## Subscribe in a Lightning Web Component

To use the `empApi` in your Lightning web component, import its methods from the `lightning/empApi` module as follows.

```
import { subscribe, unsubscribe, onError, setDebugFlag, isEmpEnabled }  
  from 'lightning/empApi';
```

Then call the imported methods in your JavaScript code.

For an example of how to use the `lightning/empApi` module and a complete reference, see the [lightning-emp-api documentation](#) in the *Lightning Component Library*.

## Subscribe in an Aura Component

To use the `empApi` methods in your Aura component, add the `lightning:empApi` component inside your custom component and assign an `aura:id` attribute to it.

```
<lightning:empApi aura:id="empApi"/>
```

Then in the client-side controller, add functions to call the `empApi` methods.

For an example of how to use the `lightning:empApi` component and a complete reference, see the [lightning:empApi documentation](#) in the *Lightning Component Library*.

## Example: Subscribe to and Replay Events Using a Visualforce Page

---

The Visualforce sample app shows you how to subscribe to durable streaming events for PushTopic and generic events. The app contains two interactive Visualforce pages: one for PushTopic events and one for generic events. You can generate test events and view them on each page. You specify which events are retrieved and displayed by setting replay options.

For each Visualforce page, the logic for replaying events is contained within a Visualforce component. The component registers the Salesforce-provided CometD extension and sets replay options.

### IN THIS SECTION:

#### [Prerequisites](#)

Set up permissions that are required to run the durable streaming samples.

#### [Deploy a Sample Project to Your Org](#)

Use the Salesforce CLI to copy all project components to your org.

#### [Durable PushTopic Streaming Sample](#)

The Durable PushTopic Streaming Visualforce sample shows you how to use replay options to subscribe and receive durable PushTopic event notifications.

#### [Durable Generic Streaming Sample](#)

The Durable Generic Streaming Visualforce sample shows you how to use replay options to subscribe and receive durable generic event notifications.

#### [Replay Events Sample: Code Walkthrough](#)

Learn how to register and use the CometD replay extension in JavaScript.

## Prerequisites

Set up permissions that are required to run the durable streaming samples.

- You must have access to a Developer Edition org and have the API Enabled and Streaming API permissions enabled. These permissions are enabled by default in a Developer Edition org, but an admin might have changed them.

To create a Developer Edition org, go to [developer.salesforce.com/signup](https://developer.salesforce.com/signup) and follow the instructions for signing up for a Developer Edition organization.

- To receive notifications, the logged-in user must have Read permission on the StreamingChannel standard object.
- To create and manage notifications, the logged-in user must have Create permission on the StreamingChannel standard object.
- To save the Apex class, the logged-in user must have the Author Apex permission.
- To save the Visualforce page, the logged-in user must have the Customize Application permission.

## Deploy a Sample Project to Your Org

Use the Salesforce CLI to copy all project components to your org.

1. Download the [Salesforce Durable Streaming Demo.zip](#) file from the *developerforce* github repository.

You can browse the contents of the project at <https://github.com/developerforce/SalesforceDurableStreamingDemo>. The sample app contains two Visualforce pages with related components and some common components. These common components are installed in your org when you deploy the .zip file.

Component	Description
<a href="#">cometdReplayExtension</a>	Static resource representing a CometD extension in JavaScript. This extension implements the replay mechanism for Streaming API.
<a href="#">cometd, jquery, jquery_cometd, json2</a>	Static resources for CometD 3.1.0, jquery, and JSON.

These app components are for the Durable PushTopic Streaming page.

Component	Description
<a href="#">DurablePushTopicEventDisplay</a>	A Visualforce component that uses the CometD extension <code>cometdReplayExtension</code> to replay events. The extension handles the handshake and subscribe calls and sets replay options.  Having the replay functionality in a Visualforce component allows you to add it to your Visualforce page for reuse in your app.
<a href="#">DurablePushTopicStreamingController</a>	Apex controller that holds the logic behind the Visualforce page.
<a href="#">DurablePushTopicStreamingDemo Visualforce Page</a>	Visualforce page. This page is the main page you use to generate, view, and replay durable PushTopic events.

These app components are for the Durable Generic Streaming page.

Component	Description
<a href="#">DurableGenericEventDisplay</a>	A Visualforce component that uses the CometD extension <code>cometdReplayExtension</code> to replay events. The extension handles the handshake and subscribe calls and sets replay options.  Having the replay functionality in a Visualforce component allows you to add it to your Visualforce page for reuse in your app.
<a href="#">DurableGenericStreamingController</a>	Apex controller that holds the logic behind the Visualforce page.
<a href="#">StreamingChannel</a>	Custom object used for creating streaming channels.
<a href="#">DurableGenericStreamingDemo Visualforce Page</a>	Visualforce page. This page is the main page you use to generate, view, and replay durable generic events.
<a href="#">DurableStreamingDemo Permission Set</a>	Permission set used to grant read and create access to the <code>StreamingChannel</code> sObject.

You use the Salesforce CLI to migrate the zip file to your org.

2. To download and install the Salesforce CLI, see [Install Salesforce CLI](#) in the *Salesforce CLI Setup Guide*.

3. To log in to your org, enter this CLI command in a Terminal window, and then log in via the web browser.

```
sf org login web
```

4. To deploy the zip file that you downloaded, enter this CLI command. Replace *<Path to zip file>* with the path to the zip that you downloaded, and replace *<Username>* with your org's username.

```
sf project deploy start --metadata-dir <Path to zip file> -o <Username>
```

After the command executes, it prints out a list of the components that were deployed to your org.

SEE ALSO:

[GitHub: Streaming Replay Client Extensions](#)

[Salesforce CLI Command Reference](#)

## Assign a Permission Set

1. From Setup, enter *Permission Sets* in the Quick Find box, then select **Permission Sets**.
2. Click **DurableStreamingDemo**, and then click **Manage Assignments**.
3. Click **Add Assignments**.
4. Click the checkbox next to the user who is running the sample, and then click **Assign**.
5. Click **Done**.

## Durable PushTopic Streaming Sample

The Durable PushTopic Streaming Visualforce sample shows you how to use replay options to subscribe and receive durable PushTopic event notifications.

### Use a Visualforce Page to Generate and Replay PushTopic Events

In this step, you use a Visualforce page to generate your own PushTopic streaming events and replay those events by using different options.

When the Visualforce page is loaded, it creates a PushTopic for the Account object. The page also subscribes to this topic to receive notifications for account creations, updates, and deletions, with an option to replay events. You can specify the name of the account to create, update, and delete on the Visualforce page. These operations generate event notifications, which are displayed in the Notifications section. You can control which events are received and displayed by subscribing with replay options. After generating events, you can replay events starting from:

- All events after a particular event specified by a replay ID.
- The first event broadcast right after subscribing (replay option -1).
- The earliest retained event in your org that's less than 24 hours old (replay option -2). The sample uses replay option -2 as the default option.

This Visualforce sample is part of the Durable Streaming Demo app.

1. From App Launcher, open the **Durable Streaming Demo** app.
2. Click the **Durable PushTopic Streaming Demo** tab.

The Visualforce page loads and subscribes to the PushTopic it created for the Account object.

- On the Visualforce page, generate some events for an account. For example, *Test account*.
- Click **Create, Update, Delete New Account**.



**Note:** The page subscribes to all new and old events by default (-2). The page first displays debug information about the CometD connection in the Notifications section followed by the events received. The first time you generate events, there are no stored events, and you see only the new events.

- To change the point in time when events are read, enter the replay ID to read from in the **Replay From Id** field. For example, to read all events after the event with replay ID 2, enter 2. Then click **Update Subscription**. The Notifications section is updated and shows only the last event with replay ID 3.

The screenshot shows a Visualforce page with three main sections:

- Replay Settings:** Contains a text field for 'Channel' with the value '/topic/TestAccountStreaming', a text field for 'Replay From Id' with the value '2', and a note '(-2 = earliest, -1 = no replay)'. Below these is an 'Update Subscription' button.
- Generate DML Events:** Contains a text field for 'New Account Name' with the value 'Test account', and a 'Create, Update, Delete New Account' button.
- Notifications:** A section titled 'Notifications' containing a list of received notifications. The first two are 'Unsubscribe Successful' and 'Subscribe Successful' messages. The third is a 'Notification on channel: "/topic/TestAccountStreaming' message with a 'Replay Id: 3' and a 'Type: "deleted"'.

- To receive only the events that are sent after you subscribe, enter *-1* in the **Replay From ID** field. Then click **Update Subscription**. The Notifications section is cleared, because only new events from this point on are shown.
- Generate some new events like you did previously using *Lightning* for the account name. The Notifications section is updated with the new events and doesn't show the old events.

▼ Replay Settings

Channel: /topic/TestAccountStreaming  
Replay From Id:   
(-2 = earliest, -1 = no replay)

▼ Generate DML Events

New Account Name:

Notifications

Received notifications should appear here...

DEBUG: Unsubscribe Successful {"clientId":"1yiss5qwe8y5t19tmf2mxkh1p9","channel":"/meta/unsubscribe","id":"11","subscription":"/topic/TestAccountStreaming","successful":true}

DEBUG: Subscribe Successful /topic/TestAccountStreaming:  
{"clientId":"1yiss5qwe8y5t19tmf2mxkh1p9","channel":"/meta/subscribe","id":"12","subscription":"/topic/TestAccountStreaming","successful":true}

Notification on channel: "/topic/TestAccountStreaming"  
Replay Id: 4  
Type: "created"  
SObject data: {"Website":null,"Id":"001D000000Ky76NIAR","Name":"Lightning"}  
Full message: {"data":{"event":{"createdDate":"2017-05-22T20:55:54.051Z","replayId":4,"type":"created"},"subject":{"Website":null,"Id":"001D000000Ky76NIAR","Name":"Lightning"},"channel":"/topic/TestAccountStreaming"}}

Notification on channel: "/topic/TestAccountStreaming"  
Replay Id: 5  
Type: "updated"  
SObject data: {"Website":null,"Id":"001D000000Ky76NIAR","Name":"Lightning\_UPDATED"}  
Full message: {"data":{"event":{"createdDate":"2017-05-22T20:55:54.421Z","replayId":5,"type":"updated"},"subject":{"Website":null,"Id":"001D000000Ky76NIAR","Name":"Lightning\_UPDATED"},"channel":"/topic/TestAccountStreaming"}}

Notification on channel: "/topic/TestAccountStreaming"  
Replay Id: 6  
Type: "deleted"  
SObject data: {"Id":"001D000000Ky76NIAR"}  
Full message: {"data":{"event":{"createdDate":"2017-05-22T20:55:54.965Z","replayId":6,"type":"deleted"},"subject":{"Id":"001D000000Ky76NIAR"},"channel":"/topic/TestAccountStreaming"}}

- Switch the replay option back to -2.  
The page displays all events, including events that were sent earlier.

```

Notifications

Received notifications should appear here...
DEBUG: Unsubscribe Successful {"clientId":"1yiss5qwe8y5t19tmf2mxkh1p9","channel":"/meta/unsubscribe","id":"15","subscription":"/topic/TestAccountStreaming","successful":true}
DEBUG: Subscribe Successful /topic/TestAccountStreaming:
{"clientId":"1yiss5qwe8y5t19tmf2mxkh1p9","channel":"/meta/subscribe","id":"16","subscription":"/topic/TestAccountStreaming","successful":true}

Notification on channel: "/topic/TestAccountStreaming"
Replay Id: 1
Type: "created"
SObject data: {"VWebsite":null,"Id":"001D000000Ky76IIAR","Name":"Test account"}
Full message: {"data":{"event":{"createdDate":"2017-05-22T20:54:06.657Z","replayId":1,"type":"created"},"subject":{"Website":null,"Id":"001D000000Ky76IIAR","Name":"Test account"},"channel":"/topic/TestAccountStreaming"}

Notification on channel: "/topic/TestAccountStreaming"
Replay Id: 2
Type: "updated"
SObject data: {"VWebsite":null,"Id":"001D000000Ky76IIAR","Name":"Test account_UPDATED"}
Full message: {"data":{"event":{"createdDate":"2017-05-22T20:54:06.490Z","replayId":2,"type":"updated"},"subject":{"Website":null,"Id":"001D000000Ky76IIAR","Name":"Test account_UPDATED"},"channel":"/topic/TestAccountStreaming"}

Notification on channel: "/topic/TestAccountStreaming"
Replay Id: 3
Type: "deleted"
SObject data: {"Id":"001D000000Ky76IIAR"}
Full message: {"data":{"event":{"createdDate":"2017-05-22T20:54:09.552Z","replayId":3,"type":"deleted"},"subject":{"Id":"001D000000Ky76IIAR"},"channel":"/topic/TestAccountStreaming"}

Notification on channel: "/topic/TestAccountStreaming"
Replay Id: 4
Type: "created"
SObject data: {"VWebsite":null,"Id":"001D000000Ky76NIAR","Name":"Lightning"}
Full message: {"data":{"event":{"createdDate":"2017-05-22T20:55:54.051Z","replayId":4,"type":"created"},"subject":{"VWebsite":null,"Id":"001D000000Ky76NIAR","Name":"Lightning"},"channel":"/topic/TestAccountStreaming"}

Notification on channel: "/topic/TestAccountStreaming"
Replay Id: 5
Type: "updated"
SObject data: {"VWebsite":null,"Id":"001D000000Ky76NIAR","Name":"Lightning_UPDATED"}
Full message: {"data":{"event":{"createdDate":"2017-05-22T20:55:54.421Z","replayId":5,"type":"updated"},"subject":{"VWebsite":null,"Id":"001D000000Ky76NIAR","Name":"Lightning_UPDATED"},"channel":"/topic/TestAccountStreaming"}

Notification on channel: "/topic/TestAccountStreaming"
Replay Id: 6
Type: "deleted"
SObject data: {"Id":"001D000000Ky76NIAR"}
Full message: {"data":{"event":{"createdDate":"2017-05-22T20:55:54.965Z","replayId":6,"type":"deleted"},"subject":{"Id":"001D000000Ky76NIAR"},"channel":"/topic/TestAccountStreaming"}

```

## Durable Generic Streaming Sample

The Durable Generic Streaming Visualforce sample shows you how to use replay options to subscribe and receive durable generic event notifications.

### Create a Streaming Channel

You must have the appropriate Streaming API permissions enabled in your org.

Create a StreamingChannel object by using the Salesforce UI.

1. Log in to your Developer Edition org.
2. If you're using Salesforce Classic, under All Tabs (+), select **Streaming Channels**. If you're using Lightning Experience, from the App Launcher, select **All Items**, and then click **Streaming Channels**.
3. On the Streaming Channels page, click **New** to create a streaming channel.
4. Enter `/u/TestStreaming` in **Streaming Channel Name** and add an optional description. Your new Streaming Channel page looks something like this:

5. Click **Save**. You now have a streaming channel that clients can subscribe to for notifications.

StreamingChannel is a regular, creatable Salesforce object, so you can also create one programmatically using Apex or a data API like the SOAP API or REST API, or using a tool such as the Developer Console. For more information, see [Reference: StreamingChannel](#).

## Use a Visualforce Page to Generate and Replay Generic Events

In this step, you use a Visualforce page to generate your own streaming events and replay those events by using different options.

The Visualforce page simulates a streaming client that subscribes to events with options to replay events. The Visualforce page allows you to supply the event's message and specify the number of messages to create. The page listens to events and displays the received events in the Notifications section. After generating events, you can replay events starting from:

- All events after a particular event specified by a replay ID.
- The first event broadcast right after subscribing (replay option -1).
- The earliest retained event in your org that's less than 24 hours old (replay option -2). The sample uses replay option -2 as the default option.

This Visualforce sample is part of the Durable Streaming Demo app.

1. From App Launcher, open the **Durable Streaming Demo** app.
2. Click the **Durable Generic Streaming Demo** tab.

The Visualforce page loads and subscribes to the test channel you created earlier.

3. In the Visualforce page, generate some events. Enter any text for the message text, for example, *Test message*. For Number of Events, enter *10*.
4. Click **Generate**.



**Note:** The page subscribes to all events by default (-2). The page first displays debug information about the CometD connection in the Notifications section followed by the events received. The first time you generate events, there are no stored events, and you see only the new events.

5. To change the point in time when events are read, enter the replay ID to read from in the **Replay From Id** field. For example, to read all events after the event with replay ID 5, enter *5*. Then click **Update Subscription**. The Notifications section is updated and shows only the last five events starting from replay ID 6.

▼ Replay Settings

Channel: /u/TestStreaming (id: 0M6D0000004CBnKAM)

Replay From Id:  (~2 = earliest, -1 = no replay)

Update Subscription

▼ Generate New Events

Message:

Number of Events:

Generate

Notifications

Received notifications should appear here...

DEBUG: Unsubscribe Successful ("clientId":"31r01442zf1gokc3g2wsc3tszl","channel":"/meta/unsubscribe","id":"9","subscription":"/u/TestStreaming","successful":true)

DEBUG: Subscribe Successful /u/TestStreaming: ("clientId":"31r01442zf1gokc3g2wsc3tszl","channel":"/meta/subscribe","id":"10","subscription":"/u/TestStreaming","successful":true)

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 6

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.065Z","replayId":6},"channel":"/u/TestStreaming"}}

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 7

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.066Z","replayId":7},"channel":"/u/TestStreaming"}}

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 8

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.066Z","replayId":8},"channel":"/u/TestStreaming"}}

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 9

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.066Z","replayId":9},"channel":"/u/TestStreaming"}}

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 10

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.066Z","replayId":10},"channel":"/u/TestStreaming"}}

- To receive only the events that are sent after you subscribe, enter `-1` in the **Replay From Id** field. Then click **Update Subscription**. The Notifications section is cleared, because only new events from this point on are shown.
- Generate some new events like you did in step 3 with `New events` for the message. The Notifications section is updated with the new events and doesn't show the old events.

▼ Replay Settings

Channel: /u/TestStreaming (id: 0M6D0000004CBnKAM)  
Replay From Id: -1 (-2 = earliest, -1 = no replay)  
Update Subscription

▼ Generate New Events

Message: New Events  
Number of Events: 10  
Generate

Notifications

Received notifications should appear here...

DEBUG: Unsubscribe Successful {"clientId":"31r01442zf1gokc3g2wsc3tszl","channel":"/meta/unsubscribe","id":"13","subscription":"/u/TestStreaming","successful":true}

DEBUG: Subscribe Successful /u/TestStreaming: {"clientId":"31r01442zf1gokc3g2wsc3tszl","channel":"/meta/subscribe","id":"14","subscription":"/u/TestStreaming","successful":true}

Notification on channel: "/u/TestStreaming"  
Payload: "New Events"  
Replay Id: 11  
Full message: {"data":{"payload":"New Events","event":{"createdDate":"2017-05-19T23:16:52.545Z","replayId":11},"channel":"/u/TestStreaming"}

Notification on channel: "/u/TestStreaming"  
Payload: "New Events"  
Replay Id: 12  
Full message: {"data":{"payload":"New Events","event":{"createdDate":"2017-05-19T23:16:52.573Z","replayId":12},"channel":"/u/TestStreaming"}

Notification on channel: "/u/TestStreaming"  
Payload: "New Events"  
Replay Id: 13  
Full message: {"data":{"payload":"New Events","event":{"createdDate":"2017-05-19T23:16:52.573Z","replayId":13},"channel":"/u/TestStreaming"}

- Switch the replay option back to -2.  
The page displays all events, including events that were sent earlier.

▼ Replay Settings

Channel: /u/TestStreaming (id: 0M6D0000004CBnKAM)

Replay From Id:  (~2 = earliest, -1 = no replay)

Update Subscription

▼ Generate New Events

Message:

Number of Events:

Generate

Notifications

Received notifications should appear here...

DEBUG: Unsubscribe Successful {"clientId":"31r01442zf1gokc3g2wsc3tszl","channel":"/meta/unsubscribe","id":"16","subscription":"/u/TestStreaming","successful":true}

DEBUG: Subscribe Successful /u/TestStreaming: {"clientId":"31r01442zf1gokc3g2wsc3tszl","channel":"/meta/subscribe","id":"17","subscription":"/u/TestStreaming","successful":true}

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 1

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.028Z","replayId":1},"channel":"/u/TestStreaming"}}

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 2

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.064Z","replayId":2},"channel":"/u/TestStreaming"}}

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 3

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.065Z","replayId":3},"channel":"/u/TestStreaming"}}

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 4

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.065Z","replayId":4},"channel":"/u/TestStreaming"}}

Notification on channel: "/u/TestStreaming"

Payload: "Test message"

Replay Id: 5

Full message: {"data":{"payload":"Test message","event":{"createdDate":"2017-05-19T23:12:20.065Z","replayId":5},"channel":"/u/TestStreaming"}}

## Replay Events Sample: Code Walkthrough

Learn how to register and use the CometD replay extension in JavaScript.

### JavaScript Sample for Replaying Events

The Visualforce components in the durable streaming sample implement a CometD client that subscribes with replay options. The components are embedded in Visualforce pages.

- For generic events, the Visualforce component is [DurableGenericEventDisplay](#).
- For PushTopic events, the Visualforce component is [DurablePushTopicEventDisplay](#).

If you want to implement a CometD client with replay options, you can reuse the Visualforce components or adapt the JavaScript code for your app. Portions of the sample component are highlighted in this section.

The first step is to register the Salesforce-provided CometD extension `cometdReplayExtension` to replay events. This snippet also sets the streaming channel and the replay option. The first argument in `registerExtension` is an arbitrary name, which you use to unregister the extension.

```
// Register Generic Streaming Replay extension
var replayExtension = new cometdReplayExtension();
```

```
replayExtension.setChannel(<Streaming Channel to Subscribe to>;
replayExtension.setReplay(<Event Replay Option>;
cometd.registerExtension('myReplayExtensionName', replayExtension);
```

Next, the client connects to the CometD replay endpoint. The API version in the endpoint must be 37.0 or later. The session ID value of the current session is passed in the `Authorization` header. The client calls the `cometd.configure()` function to set up the connection and specify the endpoint and authorization header. Next, the client performs a handshake by calling `cometd.handshake()` function.

```
// Connect to the CometD endpoint
cometd.configure({
  url: window.location.protocol+'//'+window.location.hostname+
    (null != window.location.port ? (':' + window.location.port) : '') +
    '/cometd/37.0/',
  requestHeaders: { Authorization: 'OAuth {!$Api.Session_ID}' }
});
cometd.handshake();
```

To ensure that every step in the connection process is successful before moving on to the next step, the client uses listeners. For example, a listener for the `/meta/handshake` channel checks whether the handshake is successful. If it is successful, the `subscribe()` function is called.


```
if(!metaHandshakeListener) {
  metaHandshakeListener = cometd.addListener('/meta/handshake', function(message) {
    if (message.successful) {
      $('#content').append('<br><br> DEBUG: Handshake Successful: ' +
        JSON.stringify(message) + ' <br><br>');

      if (message.ext && message.ext[REPLAY_FROM_KEY] == true) {
        isExtensionEnabled = true;
      }
      subscribedToChannel = subscribe(channel);
    } else
      $('#content').append('DEBUG: Handshake Unsuccessful: ' +
        JSON.stringify(message) + ' <br><br>');
  });
}
```

The last step is to specify a callback for the CometD `subscribe()` function. CometD calls this callback function when a message is received in the channel. In this sample, the callback function displays the message data to the page. It appends the data to the `div` HTML element whose ID value is `"content"`.

```
function subscribe(channel) {
  // Subscribe to a topic.
  // JSON-encoded update will be returned in the callback.
  return cometd.subscribe(channel, function(message) {
    var div = document.getElementById('content');
    div.innerHTML = div.innerHTML + '<p>Notification ' +
      'on channel: ' + JSON.stringify(message.channel) + '<br>' +
      'Payload: ' + JSON.stringify(message.data.payload) + '<br>' +
      'Replay Id: ' + JSON.stringify(message.data.event.replayId) + '<br>' +
      'Full message: ' + JSON.stringify(message) + '</p><br>';
  });
}
```

## cometdReplayExtension Extension

 **Note:** The extension is a prerequisite for the replay functionality in a CometD client. In the durable streaming sample, the Visualforce components register and use the extension. If you implement a CometD client, include the replay extension in your project, but you don't have to modify it.

The `cometdReplayExtension` contains cometd extension functions that are called for incoming and outgoing messages. These extension functions implement the logic that checks for the extension's registration on handshake and sets the replay option on subscription.

On handshake, the function for incoming messages checks whether the replay extension has been registered. If so, it sets the `_extensionEnabled` variable to `true`. This function also saves the replay ID of the received message so that it can be used when a client reconnects after a timeout.

```
this.incoming = function(message) {
  if (message.channel === '/meta/handshake') {
    if (message.ext && message.ext[REPLAY_FROM_KEY] == true) {
      _extensionEnabled = true;
    } else if (message.channel === _channel && message.data && message.data.event &&
               message.data.event.replayId) {
      _replay = message.data.event.replayId;
    }
  }
}
```


On subscription, the function for outgoing messages checks whether the replay extension has been registered by inspecting the `_extensionEnabled` variable. If the extension is registered, the function subscribes to events based on the specified replay option. The sample sets the replay option by calling the extension's `setReplay()` function.

```
this.outgoing = function(message) {
  if (message.channel === '/meta/subscribe') {
    if (_extensionEnabled) {
      if (!message.ext) {
        message.ext = {};
      }
      var replayFromMap = {};
      replayFromMap[_channel] = _replay;
      // add "ext : { "replay" : { CHANNEL : REPLAY_VALUE } }"
      // to subscribe message.
      message.ext[REPLAY_FROM_KEY] = replayFromMap;
    }
  }
};
```

## Example: Interactive Visualforce Page without Replay

The interactive Visualforce example shows you how to implement Streaming API from a Visualforce page. The sample uses the Dojo library and CometD to subscribe to PushTopic events.

On the Visualforce page, you enter the name of the PushTopic channel you want to subscribe to and click **Subscribe** to receive notifications on the page. Click **Unsubscribe** to unsubscribe from the channel and stop receiving notifications.

 **Note:** This sample doesn't use the replay extension and can't receive past events. To use a replay option, check out [Example: Subscribe to and Replay Events Using a Visualforce Page](#).

## IN THIS SECTION:

[Prerequisites](#)

You need access and appropriate permissions to complete the code example.

[Step 1: Create an Object](#)

Create an Invoice Statement object from the user interface.

[Step 2: Create a PushTopic](#)

Use the Developer Console to create the PushTopic record that contains a SOQL query. Event notifications are generated for updates that match the query.

[Step 3: Create the Static Resources](#)[Step 4: Create a Visualforce Page](#)[Step 5: Test the PushTopic Channel](#)

## Prerequisites

You need access and appropriate permissions to complete the code example.

- Access to a Developer Edition organization.  
To create a Developer Edition org, go to [developer.salesforce.com/signup](https://developer.salesforce.com/signup) and follow the instructions for signing up for a Developer Edition organization.
- The API Enabled permission must be enabled for your Developer Edition organization. This permission is enabled by default, but an admin might have changed it.
- The Streaming API permission must be enabled in Setup, in the User Interface page. This permission is enabled by default, but an admin might have changed it.
- The logged-in user must have Read permission on the PushTopic standard object to receive notifications.
- The logged-in user must have Create permission on the PushTopic standard object to create and manage PushTopic records.
- The logged-in user must have Author Apex permissions to create a PushTopic by using the Developer Console.

## Step 1: Create an Object

Create an Invoice Statement object from the user interface.

After you create a PushTopic and subscribe to it, you'll get notifications when an Invoice Statement record is created, updated, deleted, or undeleted. You'll create the object with the user interface.

1. From your management settings for custom objects, if you're using Salesforce Classic, click **New Custom Object**, or if you're using Lightning Experience, select **Create > Custom Object**.
2. Define the custom object.
  - In the **Label field**, type *Invoice Statement*.
  - In the **Plural Label field**, type *Invoice Statements*.
  - Select **Starts with vowel sound**.
  - In the **Record Name field**, type *Invoice Number*.
  - In the **Data Type field**, select *Auto Number*.
  - In the **Display Format field**, type *INV- {0000}*.
  - In the **Starting Number field**, type *1*.

3. Click **Save**.
4. Add a Status field.
  - a. Scroll down to the Custom Fields & Relationships related list and click **New**.
  - b. For Data Type, select `Picklist` and click **Next**.
  - c. In the Field Label field, type `Status`.
  - d. Type the following picklist values in the box provided, with each entry on its own line.

```
Open
Closed
Negotiating
Pending
```

- e. Select the checkbox for **Use first value as default value**.
  - f. Click **Next**.
  - g. For field-level security, select `Read Only` and then click **Next**.
  - h. Click **Save & New** to save this field and create a new one.
5. Now create an optional Description field.
  - a. In the Data Type field, select `Text Area` and click **Next**.
  - b. In the Field Label and Field Name fields, enter `Description`.
  - c. Click **Next**, accept the defaults, and click **Next** again.
  - d. Click **Save** to go the detail page for the Invoice Statement object.

Your Invoice Statement object should now have two custom fields.

## Step 2: Create a PushTopic

Use the Developer Console to create the PushTopic record that contains a SOQL query. Event notifications are generated for updates that match the query.

1. Open the Developer Console.
2. Click **Debug > Open Execute Anonymous Window**.
3. In the Enter Apex Code window, paste in the following Apex code, and click **Execute**.

```
PushTopic pushTopic = new PushTopic();
pushTopic.Name = 'InvoiceStatementUpdates';
pushTopic.Query = 'SELECT Id, Name, Status__c, Description__c FROM Invoice_Statement__c';
pushTopic.ApiVersion = 63.0;
pushTopic.NotifyForOperationCreate = true;
pushTopic.NotifyForOperationUpdate = true;
pushTopic.NotifyForOperationUndelete = true;
pushTopic.NotifyForOperationDelete = true;
pushTopic.NotifyForFields = 'Referenced';
insert pushTopic;
```



**Note:** If your organization has a namespace prefix defined, then you'll need to preface the custom object and field names with that namespace when you define the PushTopic query. For example, `SELECT Id, Name,`

```
namespace__ Status__c, namespace__ Description__c FROM
namespace__ Invoice_Statement__c.
```

Because `NotifyForOperationCreate`, `NotifyForOperationUpdate`, `NotifyForOperationDelete` and `NotifyForOperationUndelete` are set to `true`, Streaming API evaluates records that are created, updated, deleted, or undeleted and generates a notification if the record matches the `PushTopic` query. Because `NotifyForFields` is set to `Referenced`, Streaming API will use fields in both the `SELECT` clause and the `WHERE` clause to generate a notification. Whenever the fields `Name`, `Status__c`, or `Description__c` are updated, a notification will be generated on this channel. For more information about `NotifyForOperationCreate`, `NotifyForOperationUpdate`, `NotifyForOperationDelete`, `NotifyForOperationUndelete`, and `NotifyForFields`, see [Event Notification Rules](#).



**Note:** In API version 28.0 and earlier, notifications are only generated when records are created or updated. The `NotifyForOperationCreate`, `NotifyForOperationUpdate`, `NotifyForOperationDelete`, and `NotifyForOperationUndelete` fields are unavailable and the `NotifyForOperations` enum field is used instead to set which record events generate a notification. For more information see [PushTopic](#).

## Step 3: Create the Static Resources

1. Download this static resource .zip file: [streaming\\_api\\_interactive\\_visualforce\\_demo-v40.zip](#)
2. Extract the following files from the .zip file:

File Name	Description
cometd.zip	Files for CometD version 3.1.0 and the Dojo toolkit used by <code>demo.js</code> . When you define a .zip archive file as a static resource, Visualforce can access the files in that archive. The .zip file becomes a virtual file system.
demo.css	The CSS code that formats the Visualforce page.
demo.js	The code used by the page to subscribe to the channel, receive and display the notifications, and unsubscribe from the channel.
json2.js	The JavaScript library that contains the <code>stringify</code> and <code>parse</code> methods.
StreamingApiDemo	The Visualforce page that displays the Streaming API notifications.

3. From Setup, enter *Static Resources* in the Quick Find box, then select **Static Resources** to add the extracted files with the following names:

File Name	Static Resource Name
cometd.zip	<i>cometd_zip</i>
demo.css	<i>demo_css</i>
demo.js	<i>demo_js</i>
json2.js	<i>json2_js</i>

## Step 4: Create a Visualforce Page

Create a Visualforce page to display the channel notifications.

1. From Setup, enter *Visualforce Pages* in the Quick Find box, then select **Visualforce Pages**.
2. Click **New**.
3. In the **Label** field, enter the name of the page *StreamingAPIDemo*.
4. Replace the code in the page with the code from the *StreamingApiDemo* file that you downloaded.

```
<apex:page >
<apex:includeScript value="{!$Resource.json2_js}"/>
<script type="text/javascript" src="{!URLFOR($Resource.cometd_zip, 'dojo/dojo.js')}"/>
data-dojo-config="async: 1"></script>
<apex:stylesheet value="{!$Resource.demo_css}"/>
<script>var token = '{!$Api.Session_ID}';</script>
    <div id="demo">
        <div id="datastream"></div>
        <script type="text/javascript" src="{!$Resource.demo_js}">
</script>

        <div id="input">
            <div id="join">
                <table>
                    <tbody>
                        <tr>
                            <td>&nbsp;</td>
                            <td> Enter Topic Name </td>
                            <td>
                                <input id="topic" type="text" />
                            </td>
                            <td>
                                <button id="subscribeButton"
                                    class="button">Subscribe</button>
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
            <div id="joined">
                <table>
                    <tbody>
                        <tr>
                            <td>
                                <button id="leaveButton"
                                    class="button">Unsubscribe</button>
                            </td>
                        </tr>
                    </tbody>
                </table>
            </div>
        </div>
    </div>
</apex:page>
```

5. Click **Save** to save the page.

## Step 5: Test the PushTopic Channel

1. To load the Visualforce page in a web browser, click **Preview** on the Visualforce page editor:
2. In the text box, enter the channel name: `/topic/InvoiceStatementUpdates`.
3. To subscribe to the channel, click **Subscribe**.
4. Create or modify an invoice statement in a different browser tab. The page displays some debug messages and event notifications. The output resembles the following:

```
DEBUG: Handshake Successful: {
  "ext":{"replay":true,"payload.format":true},
  "minimumVersion":"1.0",
  "clientId":"41kdiuvgig2tfxdh9weakuiwyh",
  "supportedConnectionTypes":["long-polling"],
  "channel":"/meta/handshake",
  "id":"1","version":"1.0","successful":true,"reestablish":false}

DEBUG: Connection Successful : {
  "clientId":"41kdiuvgig2tfxdh9weakuiwyh",
  "advice":{"
    "interval":2000,"multiple-clients":true,"reconnect":"retry",
    "timeout":110000},"channel":"/meta/connect","id":"2",
    "successful":true}

DEBUG: Subscribe Successful /topic/InvoiceStatementUpdates:
{"clientId":"41kdiuvgig2tfxdh9weakuiwyh","channel":"/meta/subscribe",
  "id":"15","subscription":"/topic/InvoiceStatementUpdates","successful":true}

{
  "event": {
    "createdDate": "2017-05-16T23:05:50.173Z",
    "replayId": 1,
    "type": "created"
  },
  "sobject": {
    "Description__c": "New invoice.",
    "Id": "a00D0000009YbwQIAS",
    "Status__c": "Open",
    "Name": "INV-0001"
  }
}

_____

{
  "event": {
    "createdDate": "2017-05-16T23:06:11.529Z",
    "replayId": 2,
    "type": "updated"
  },
  "sobject": {
```

```

    "Description__c": "New invoice.",
    "Id": "a00D00000009YbwQIAS",
    "Status__c": "Negotiating",
    "Name": "INV-0001"
  }
}

```

The debug messages contain information about the subscription status. The first event notification shows the notification data when an invoice statement is created. The second notification shows the notification data when an invoice statement is updated.

Click **Unsubscribe** to unsubscribe from the channel and stop receiving notifications.

## Example: Authentication

You can set up a simple authentication scheme for developer testing. For production systems, use robust authorization, such as OAuth 2.0.


### IN THIS SECTION:

[Set Up Authentication for Developer Testing](#)

[Set Up Authorization with OAuth 2.0](#)

## Set Up Authentication for Developer Testing

To set up authorization for developer testing:

 **Important:** This authorization method is simple to use and recommended for testing your code quickly. However, we recommend that you use OAuth 2.0 in a production environment for more robust security. The OAuth authentication method with a connected app provides restricted access and other benefits.

1. Log in using the SOAP API `login()` and get the session ID.
2. Set up the HTTP authorization header using this session ID:

```
Authorization: Bearer sessionId
```

The CometD endpoint requires a session ID on all requests, plus any additional cookies set by the Salesforce server.

For more information, see [Step 4: Use the Connector with Username and Password Login](#).

## Set Up Authorization with OAuth 2.0

Setting up OAuth 2.0 requires some configuration in the user interface and in other locations. If any of the steps are unfamiliar, you can consult the [REST API Developer Guide](#) or [OAuth 2.0 documentation](#).

The sample Java code in this chapter uses the Apache HttpClient library, which can be downloaded from <http://hc.apache.org/httpcomponents-client-ga/>.

1. In Salesforce Classic, from Setup, enter *Apps* in the *Quick Find* box, then select **Apps**. Or in Lightning Experience, enter *App* in the *Quick Find* box, then select **App Manager**. Click **New** in the Connected Apps related list to create a new connected app.

The `Callback URL` you supply here is the same as your Web application's callback URL. If you work with Java, it's usually a servlet. It must be secure: `http://` doesn't work, only `https://`. For development environments, the callback URL is similar to `https://my-website/_callback`.

When you save the connected app, its detail page is displayed. To view the consumer key and consumer secret, click **Manage Consumer Details**, and then verify your identity.



**Note:** The OAuth 2.0 specification uses "client" instead of "consumer." Salesforce supports OAuth 2.0.

The values here correspond to the following values in the sample code in the rest of this procedure:

- `client_id` is the Consumer Key
- `client_secret` is the Consumer Secret
- `redirect_uri` is the Callback URL.

An additional value you must specify is the `grant_type`. For OAuth 2.0 callbacks, the value is `authorization_code` as shown in the sample. For more information about these parameters, see [Authorize Apps with OAuth](#) in *Salesforce Help*.

If the value of `client_id` (or `consumer key`) and `client_secret` (or `consumer secret`) are valid, Salesforce sends a callback to the URI specified in `redirect_uri` that contains a value for `access_token`.

2. From your Java or other client application, make a request to the authentication URL that passes in `grant_type`, `client_id`, `client_secret`, `username`, and `password`. For example:

```
HttpClient httpClient = new DefaultHttpClient();
HttpPost post = new HttpPost(baseUrl);

List<BasicNameValuePair> parametersBody = new ArrayList<BasicNameValuePair>();

parametersBody.add(new BasicNameValuePair("grant_type", password));
parametersBody.add(new BasicNameValuePair("client_id", clientId));
parametersBody.add(new BasicNameValuePair("client_secret", client_secret));
parametersBody.add(new BasicNameValuePair("username", "auser@example.com"));
parametersBody.add(new BasicNameValuePair("password", "swordfish"));
```



**Important:** Use the username-password authorization flow only if you're handling your own credentials. Review the recommendations and restrictions for this authorization flow in [OAuth 2.0 Username-Password Flow for Special Scenarios](#) in *Salesforce Help*.



**Example:** This example gets the session ID (authenticates), and then follows a resource, `https://MyDomainName.my.salesforce.com/id/00Dxxxxxxxxxxxxx/005xxxxxxxxxxxxx` contained in the first response to get more information about the user.

```
public static void oAuthSessionProvider(String loginHost, String username,
    String password, String clientId, String secret)
    throws HttpException, IOException
{
    // Set up an HTTP client that makes a connection to REST API.
    DefaultHttpClient client = new DefaultHttpClient();
    HttpParams params = client.getParams();
    HttpClientParams.setCookiePolicy(params, CookiePolicy.RFC_2109);
    params.setParameter(HttpConnectionParams.CONNECTION_TIMEOUT, 30000);

    // Set the SID.
    System.out.println("Logging in as " + username + " in environment " + loginHost);
```

```

String baseUrl = loginHost + "/services/oauth2/token";
// Send a post request to the OAuth URL.
HttpPost oauthPost = new HttpPost(baseUrl);
// The request body must contain these 5 values.
List<BasicNameValuePair> parametersBody = new ArrayList<BasicNameValuePair>();
parametersBody.add(new BasicNameValuePair("grant_type", "password"));
parametersBody.add(new BasicNameValuePair("username", username));
parametersBody.add(new BasicNameValuePair("password", password));
parametersBody.add(new BasicNameValuePair("client_id", clientId));
parametersBody.add(new BasicNameValuePair("client_secret", secret));
oauthPost.setEntity(new UrlEncodedFormEntity(parametersBody, HTTP.UTF_8));

// Execute the request.
System.out.println("POST " + baseUrl + "...\\n");
HttpResponse response = client.execute(oauthPost);
int code = response.getStatusLine().getStatusCode();
Map<String, String> oauthLoginResponse = (Map<String, String>)
    JSON.parse(EntityUtils.toString(response.getEntity()));
System.out.println("OAuth login response");
for (Map.Entry<String, String> entry : oauthLoginResponse.entrySet())
{
    System.out.println(String.format(" %s = %s", entry.getKey(), entry.getValue()));
}
System.out.println("");

// Get user info.
String userIdEndpoint = oauthLoginResponse.get("id");
String accessToken = oauthLoginResponse.get("access_token");
List<BasicNameValuePair> qsList = new ArrayList<BasicNameValuePair>();
qsList.add(new BasicNameValuePair("oauth_token", accessToken));
String queryString = URLEncodedUtils.format(qsList, HTTP.UTF_8);
HttpGet userInfoRequest = new HttpGet(userIdEndpoint + "?" + queryString);
HttpResponse userInfoResponse = client.execute(userInfoRequest);
Map<String, Object> userInfo = (Map<String, Object>)
    JSON.parse(EntityUtils.toString(userInfoResponse.getEntity()));
System.out.println("User info response");
for (Map.Entry<String, Object> entry : userInfo.entrySet())
{
    System.out.println(String.format(" %s = %s", entry.getKey(), entry.getValue()));
}
System.out.println("");

// Use the user info in interesting ways.
System.out.println("Username is " + userInfo.get("username"));
System.out.println("User's email is " + userInfo.get("email"));
Map<String, String> urls = (Map<String, String>)userInfo.get("urls");
System.out.println("REST API url is " + urls.get("rest").replace("{version}",
"63.0"));
}

```

The output from this code resembles the following.

```

Logging in as auser@example.com in environment https://MyDomainName.my.salesforce.com
POST https://MyDomainName.my.salesforce.com/services/oauth2/token...

OAuth login response
  id = https://MyDomainName.my.salesforce.com/id/00D30000000ehjIEAQ/00530000003THy8AAG
  issued_at = 1334961666037
  instance_url = https://MyDomainName.my.salesforce.com
  access_token =
00D30000000ehjI!ARYAQHc.0Mlmz.DCg3HRNF.SmsSn5njPkry2SM6pb6rjCOqfAODaUkv5CGksRSPRb.xb
  signature = 8M9VWBoaEk+Bs//yD+BfrUR/+5tkNLgXAIwall1PMwsY=

User info response
  user_type = STANDARD
  status = {created_date=2012-04-08T16:44:58.000+0000, body=Hello}
  urls =
{subjects=https://MyDomainName.my.salesforce.com/services/data/v{version}/sobjects/,
feeds=https://MyDomainName.my.salesforce.com/services/data/v{version}/chatter/feeds,
users=https://MyDomainName.my.salesforce.com/services/data/v{version}/chatter/users,
query=https://MyDomainName.my.salesforce.com/services/data/v{version}/query/,
enterprise=https://MyDomainName.my.salesforce.com/services/Soap/c/{version}/00D30000000ehjI,
recent=https://MyDomainName.my.salesforce.com/services/data/v{version}/recent/,
feed_items=https://MyDomainName.my.salesforce.com/services/data/v{version}/chatter/feed-items,
search=https://MyDomainName.my.salesforce.com/services/data/v{version}/search/,
partner=https://MyDomainName.my.salesforce.com/services/Soap/u/{version}/00D30000000ehjI,
rest=https://MyDomainName.my.salesforce.com/services/data/v{version}/,
groups=https://MyDomainName.my.salesforce.com/services/data/v{version}/chatter/groups,
metadata=https://MyDomainName.my.salesforce.com/services/Soap/m/{version}/00D30000000ehjI,
profile=https://MyDomainName.my.salesforce.com/00530000003THy8AAG}
  locale = en_US
  asserted_user = true
  id = https://login.salesforce.com/id/00D30000000ehjIEAQ/00530000003THy8AAG
  nick_name = SampleNickname
  photos = {picture=https://MyDomainName.file.force.com/profilephoto/005/F,
thumbnail=https://MyDomainName.file.force.com/profilephoto/005/T}
  display_name = Sample User
  first_name = Admin
  last_modified_date = 2012-04-19T04:35:29.000+0000
  username = auser@example.com
  email = emailaddr@example.com
  organization_id = 00D30000000ehjIEAQ
  last_name = User
  utcOffset = -28800000
  active = true
  user_id = 00530000003THy8AAG
  language = en_US

Username is auser@example.com
User's email is emailaddr@example.com
REST API url is https://MyDomainName.my.salesforce.com/services/data/v63.0/

```

## CHAPTER 3 Client Connection Considerations

### In this chapter ...

- Clients and Timeouts
- Clients and Cookies for Streaming API
- Supported CometD Versions
- Streaming API Calls Blocked by Security Settings
- Debugging Streaming API Applications
- Using an Experience Cloud Site with Streaming API-Based Features

Keep in mind these client and troubleshooting considerations when implementing a Streaming API client.

## Clients and Timeouts

---

Streaming API imposes two timeouts, as supported in the Bayeux protocol.

### Socket timeout: 110 seconds

A client receives events (JSON-formatted HTTP responses) while it waits on a connection. If no events are generated and the client is still waiting, the connection times out after 110 seconds and the client closes the connection. Clients should reconnect before two minutes to avoid the Bayeux session timeout.

### Reconnect timeout: 40 seconds

After receiving the events, a client needs to reconnect to receive the next set of events. If the reconnection doesn't happen within 40 seconds, the server expires the subscription and the connection is closed. If this happens, the client must start again and handshake, subscribe, and connect.

Each Streaming API client logs into an instance and maintains a session. When the client handshakes, connects, or subscribes, the session timeout is restarted. A client session times out if the client doesn't reconnect to the server within 40 seconds after receiving a response (an event, subscribe result, and so on).

Note that these timeouts apply to the Streaming API client session and not the Salesforce authentication session. If the client session times out, the authentication session remains active until the organization-specific timeout policy goes into effect.



**Note:** In addition to timeouts, a client might disconnect from the channel due to network failures. For more information, see [Short Network Failures](#) and [Long Network Failures or Server Failures](#) in the [CometD Reference Documentation](#).

## Clients and Cookies for Streaming API

---

The client you create to work with the Streaming API must obey the standard cookie protocol with the server. The client must accept and send the appropriate cookies for the domain and URI path, for example `https://MyDomainName.my.salesforce.com/cometd`.



**Note:** To ensure continuity during instance refreshes and org migrations, we recommend using your My Domain login URL with Streaming API.

Streaming API requirements on clients:

- If the content of the post is JSON, the `"Content-Type: application/json"` header is required on all calls to the cometd servlet.
- A header containing the Salesforce session ID or OAuth token is required. For example, `Authorization: Bearer sessionId`.
- The client must accept and send back all appropriate cookies for the domain and URI path. Clients must obey the standard cookie protocol with the server.
- The subscribe response and other responses can contain the following fields. These fields aren't contained in the CometD specification.
  - `EventType` contains either `created` or `updated`.
  - `CreatedDate` contains the event's creation date.

## Supported CometD Versions

---

Use CometD version 3.1.0 or later in your clients to connect to Streaming API. Earlier versions aren't supported and could result in unexpected behavior. To prevent potential issues with old CometD versions in your clients, upgrade the CometD library to a supported version. For more information, see <https://cometd.org/>.

## Streaming API Calls Blocked by Security Settings

---

Streaming API calls can be blocked if the session setting for locking sessions is enabled.

Streaming API calls are blocked when both of these conditions apply.

- The **Lock sessions to the IP address from which they originated** session setting is enabled in Setup. See [Modify Session Security Settings](#) in *Salesforce Help*.
- The Streaming API client is using a session ID that was obtained from an IP address that's different than the client's IP address.

To resolve this issue, make sure that the IP address from where the session originated matches the client's IP address. Alternatively, disable the setting for locking sessions.

## Debugging Streaming API Applications

---

You must be able to see all of the requests and responses to debug Streaming API applications. Because Streaming API applications are stateful, you need to use a proxy tool to debug your application. Use a tool that can report the contents of all requests and results, such as [Burp Proxy](#), [Fiddler](#), or [Firebug](#).

The most common errors include:

- Browser and JavaScript issues
- Sending requests out of sequence
- Malformed requests that don't follow the Bayeux protocol
- Authorization issues
- Network or firewall issues with long-lived connections


Using these tools, you can look at the requests, headers, body of the post, as well as the results. If you must contact us for help, be sure to copy and save these elements to assist in troubleshooting.

## Handling Streaming API Errors

Learn about some common errors and how to handle them in your streaming client.

### 401 Authentication Errors

Client authentication can sometimes become invalid, for example, when the OAuth token is revoked or a Salesforce admin revokes the Salesforce session. An admin can revoke an OAuth token or delete a Salesforce session to prevent a client from receiving events. Sometimes a client can inadvertently invalidate its authentication by logging out from a Salesforce session. Streaming API regularly validates the OAuth token or session ID while the client is connected. If client authentication isn't valid, the client is notified with an error. A Bayeux message is sent on the `/meta/connect` channel with an error value of `401::Authentication invalid` and an advice field containing `reconnect=none`. After receiving the error notification in the channel listener, the client must reauthenticate and reconnect to receive new events.

 **Note:** If the OAuth or session token isn't sent in the request header, the 401 error message text is `401::Request requires authentication`.

The error response message that is sent on the `/meta/connect` channel looks similar to the following.

```
{
  "clientId": "1qlib66fvm7kli1gfoauu95i78g",
```

```

"advice": {
  "reconnect": "none",
  "interval": 0
},
"channel": "/meta/connect",
"id": 7,
"error": "401::Authentication invalid",
"successful": false
}

```

If the client is required to perform a new handshake request due to a failed connection, the authentication error is sent on the `/meta/handshake` channel. The handshake request fails with a `403::Handshake denied` error in the response. The `401::Authentication invalid` error is nested in the `ext` property in the response.

The error response message that is sent on the `/meta/handshake` channel looks similar to the following.

```

{
  "ext": {
    "sfdc": {
      "failureReason": "401::Authentication invalid"
    }
  },
  "advice": {
    "reconnect": "none"
  },
  "channel": "/meta/handshake",
  "error": "403::Handshake denied",
  "successful": false
}

```

For a code example about reauthentication, see the [AuthFailureListener class](#) in the EMPCConnector GitHub project.



**Note:** Invalidated client authentication doesn't include Salesforce session expiration. The Salesforce session never expires in a CometD client. Salesforce keeps extending the timeout interval as long as the client stays connected.

## 403 Unknown Client Error

If a long-lived connection is lost due to unexpected network disruption, the CometD server times out the client and deletes the client state. The CometD client attempts to reconnect but the connection is rejected with the `403::Unknown client` error because the client state doesn't exist anymore. The error response returned when the client attempts to reconnect after a timeout looks similar to the following message.

```

{
  "error": "403::Unknown client",
  "successful": false,
  "advice": {"interval": 0, "reconnect": "handshake"}
}

```

When the client receives the `403::Unknown client` error with the `"reconnect": "handshake"` advice field, the client must perform a new handshake. If the handshake is successful, the client must resubscribe to the channel in the handshake listener.

For more information, see [Clients and Timeouts](#).



**Note:** The `403::Unknown client` error is sometimes returned when using more than one CometD connection. You can have only one CometD connection in one browser. If you have more than one connection because you have multiple clients or

another app is using one CometD connection, your client fails to connect. In this event, ensure to turn off the other client or share the CometD connection between clients.


## 403 Resource Limit and Validation Errors for Handshake Requests

After a client sends a handshake request, Streaming API checks the client's API version and resource limits to ensure that the client can perform a successful connection. The following validations are performed.

- API Version
- Maximum concurrent clients (subscribers) across all streaming channels
- Simultaneous connections limit on the Salesforce app servers. This limit protects against denial of service attacks.

If the client fails the validation, the response contains `403::Handshake denied` in the `error` field, and the cause of the error is returned in the nested `ext/sfdc/failureReason` field. For example, the following response message is returned when the number of simultaneous connections has been exhausted.

```
{
  "channel" : "/meta/handshake",
  "id" : "1",
  "error" : "403::Handshake denied",
  "successful" : false,
  "advice" : {
    "reconnect" : "none"
  },
  "ext" : {
    "sfdc" : {
      "failureReason" : "403::To protect all customers from excessive use and Denial of
        Service attacks, we limit the number of simultaneous connections per server.
        Your request has been denied because this limit has been exceeded.
        Please try your request again later."
    },
    "replay" : true,
    "payload.format" : true
  }
}
```

 **Note:** The maximum daily event usage is checked when the client subscribes.

## 503 Server Too Busy Error

If the Salesforce servers don't have available resources to process your Streaming API request, a 503 error is returned in the `ext/sfdc/failureReason` field. This error is returned for a handshake or a connection request. For example, this response shows the 503 error on the `/meta/connect` channel.

```
{
  "channel" : "/meta/connect",
  "id" : "6",
  "error" : "401::Authentication invalid",
  "successful" : false,
  "ext" : {
    "sfdc" : {
      "failureReason" : "503::Server is too busy. Please try your request again later."
    }
  }
}
```

```

    },
    "clientId" : "blunwa43ckd43m16v60gs6v2yv7",
    "advice" : {
        "reconnect" : "none",
        "interval" : 0
    }
}

```

If you get the 503 error, try your request after a small delay of a few minutes. This error is temporary and your request will be successful when the Salesforce app servers are available again.

## Streaming API Error Codes

Learn about the errors that Streaming API can return to troubleshoot your streaming client.

Error Code	Error Message	Error Description
400	API version in the URI is mandatory. URI format: '/cometd/63.0'	The API version information isn't in the URI. Include the API version at the end of the URI. For example, /cometd/63.0.
400	Unsupported API version. Only API versions '37.0' and later are supported. URI format: '/cometd/63.0'	The supplied API version in the URI is not supported. Only API version 37.0 and later are supported. The URI format is /cometd/xx.x.
400	Invalid connection type { <i>connection_type</i> }	An invalid transport type was used. Only long-polling is supported, but another connection type was requested, such as WebSocket or callback long-polling.
400	The channel you requested to subscribe to doesn't exist { <i>channel_name</i> }	The streaming channel requested to subscribe to doesn't exist. Ensure that the channel is created before subscribing.
400	Channel name not specified	The channel name wasn't specified. Specify a valid channel name to subscribe to.
400	Channel subscriptions must start with a leading '/'	The channel name format is invalid. Channel names must start with a leading slash (/).
400	Query fields { <i>query_fields</i> } do not exist on the topic entity	The supplied query fields don't exist on the Salesforce object specified in the PushTopic.
400	Client <i>client_name</i> has established a session, but no <i>cookie_name</i> cookie present	No cookie was found after the client established a session. Ensure that the streaming client accepts cookies.
400	The replayId { <i>replay_id</i> } you provided was invalid. Please provide a valid ID, -2 to replay all events, or -1 to replay only new events.	The supplied replay ID is invalid. Ensure that the replay ID corresponds to an event that is within the retention window and that has not been deleted. Alternatively, provide -2 to replay all events or -1 to replay only new events.
400	Authenticated user id does not match the session's user id	The CometD session in the browser is associated with more than one user. This error can occur if you're using the <a href="#">Lightning Emp API</a> component and logged in as

Error Code	Error Message	Error Description
		another user using system admin login as feature. See <a href="#">Log In as Another User</a> in <i>Salesforce Help</i> . For more details, see <a href="#">this knowledge article</a> .
401	Authentication invalid.	The supplied authentication token or session ID is not valid. This error is returned on the <code>/meta/handshake</code> or the <code>/meta/connect</code> channel. On the <code>/meta/handshake</code> channel, the error is in the <code>failureReason</code> response field, which is nested under the <code>ext/sfdc</code> field. On the <code>/meta/connect</code> channel, the error is in the <code>error</code> field.
401	Request requires authentication.	No authentication token or session ID was supplied in the request header. The client must send authentication information. This error is returned in the handshake error response (on the <code>/meta/handshake</code> channel) in the <code>failureReason</code> response field, which is nested under the <code>ext/sfdc</code> field. The <code>error</code> field in the response also contains the following error: <code>403::Handshake denied.</code>
403	Cannot create channel { <i>channel_name</i> }	The subscription channel can't be created, which can be due to insufficient permissions.
403	Subscriber does not have access to the entity in this topic	The subscriber doesn't have access to the Salesforce object in the PushTopic.
403	Subscriber does not have access to all fields referenced in the where clause of the PushTopic	The subscriber doesn't have access to all fields referenced in the WHERE clause of the PushTopic.
403	Handshake denied	The handshake request was denied. The cause of this error is provided in the <code>failureReason</code> field in the response, which is nested under the <code>ext/sfdc</code> field.
403	Client has not completed handshake	The client has not completed a handshake. The client must perform a handshake before subscribing.
403	Organization concurrent user limit exceeded	The maximum number of concurrent clients across all channels has been exceeded. This error applies to any type of event, including PushTopic, generic, platform, and Change Data Capture events.
403	Organization total events daily limit exceeded	The maximum number of delivered event notifications within a 24-hour period to all CometD clients has been exceeded. This error applies to any type of event, including PushTopic, generic, platform, and Change Data Capture events.
403	Restricted channel	The user doesn't have the required permissions to subscribe to the streaming channel.
403	User not enabled for streaming	The user doesn't have read permission on the PushTopic.

Error Code	Error Message	Error Description
403	User not allowed to subscribe CDC without View All Data permissions	The user must have the View All Data permission to subscribe to Change Data Capture.
403	Subscription limit exceeded for this topic	The maximum number of concurrent clients per topic for PushTopic and generic events has been exceeded. This error doesn't apply to platform events.
403	Unknown client	The server deleted the client CometD session due to a timeout, which can be caused by a network failure or by Hyperforce's auto-scaling processes as part of elastic computing. The client must perform a new handshake and reconnect.
403	To protect all customers from excessive use and Denial of Service attacks, we limit the number of simultaneous connections per server. Your request has been denied because this limit has been exceeded. Please try your request again later.	Salesforce app servers enforce a limit on simultaneous connections per server to protect from excessive use and denial of service attacks. Your request has been denied because this limit has been exceeded. Try your request again later. This error is returned in a handshake response (on the <code>/meta/handshake</code> channel) in the <code>failureReason</code> response field, which is nested under the <code>ext/sfdc</code> field. The response also contains an error in the <code>error</code> field: <code>403::Handshake denied</code> .
413	Maximum Request Size Exceeded	The maximum request size of 32,768 bytes has been exceeded.
503	Server is too busy. Please try your request again later.	The server can't process your request because it is too busy. Try your request again later. This error is returned in the <code>failureReason</code> response field, which is nested in the <code>ext/sfdc</code> field on the <code>/meta/handshake</code> or <code>/meta/connect</code> channels. The <code>error</code> field in the response contains another error alongside this error.

## Generic Streaming-only Errors

The following errors are returned for generic streaming events only.

Error Code	Error Message	Error Description
403	Unable to create channel dynamically, maximum channel limit has been exceeded	The maximum number of generic streaming channels has been exceeded.
403	No access on channel	The generic streaming channel can't be accessed because the user doesn't have permissions on the StreamingChannel object.
404	channel names may not vary only by case	The generic streaming channel exists with a different case. Generic streaming channel names are case-sensitive.

Error Code	Error Message	Error Description
404	Unknown channel	The generic streaming channel isn't found or can't be created dynamically.

## Using an Experience Cloud Site with Streaming API-Based Features

---


If you use an Experience Cloud site and enable features that use Streaming API in the background, such as in-app or push notifications, HTTP POST requests are sent to the `/cometd/` endpoint. Sending many CometD requests can be problematic sometimes and can cause hitting HTTP request rate limits. If your Experience Cloud site is hosted on a third-party Content Delivery Network (CDN), the HTTP POST requests will be sent to the CDN. The volume of requests depends on the number of users that are simultaneously logged in to the site. Each logged-in user gets their own Streaming API session so the more users are logged in, the more POST requests are generated.

## CHAPTER 4 PushTopic Events (Legacy)

### In this chapter ...

- [Working with PushTopics](#)
- [PushTopic Considerations](#)
- [PushTopic Streaming Allocations](#)
- [Reference: PushTopic](#)

PushTopic events provide a secure and scalable way to receive notifications for changes to Salesforce data that match a SOQL query you define.

 **Important:** PushTopic events is a legacy product. Salesforce no longer enhances PushTopic events with new features and provides limited support for this product. Instead of PushTopic events, consider using Change Data Capture events. To learn about Change Data Capture events, see the [Change Data Capture Developer Guide](#) and the [Change Data Capture Basics](#) Trailhead module.

Use PushTopic events to:

- Receive notifications of Salesforce record changes, including create, update, delete, and undelete operations.
- Capture changes for the fields and records that match a SOQL query.
- Receive change notifications for only the records a user has access to based on sharing rules.
- Limit the stream of events to only those events that match a subscription filter.

A typical application of PushTopic events is refreshing the UI of a custom app from Salesforce record changes.

PushTopic events can be received by:

- Pages in the Salesforce application
- Application servers outside of Salesforce
- Clients outside the Salesforce application

The generation of PushTopic notifications follows this sequence.

1. Create a PushTopic based on a SOQL query. The PushTopic defines the channel.
2. Clients subscribe to the channel.
3. A record is created, updated, deleted, or undeleted (an event occurs). The changes to that record are evaluated.
4. If the record changes match the criteria of the PushTopic query, a notification is generated by the server and received by the subscribed clients.

## Working with PushTopics

Each PushTopic record that you create corresponds to a channel in CometD. The channel name is the name of the PushTopic prefixed with `/topic/`, for example, `/topic/MyPushTopic`. A Bayeux client can receive streamed events on this channel. The channel name is case-sensitive when you subscribe.

As soon as a PushTopic record is created, the system starts evaluating record creates, updates, deletes, and undeletes for matches. Whenever there's a match, a new notification is generated. The server polls for new notifications for currently subscribed channels every second. This frequency can fluctuate depending on the overall server load.



### Note:

- Record changes that Bulk API performs don't generate notifications because a large volume of changes can flood a channel. Also, changes made by tools that use Bulk API, such as the Mass Transfer tool or Data Loader, don't generate notifications.
- Cascade-deleted records don't trigger PushTopic notifications. To receive notifications for cascade-deleted records, use Change Data Capture instead. For more information, see [Change Data Capture Developer Guide](#).

The PushTopic defines when notifications are generated in the channel. Specify the conditions in these PushTopic fields:

- [PushTopic Queries](#)
- [Events](#)
- [Notifications](#)

To receive notifications, users must have read access on the object in the PushTopic query and the PushTopic itself.



**Note:** PushTopic isn't available for users in Experience Cloud sites, including Partner Community and Customer Community users. Also, PushTopic isn't available in legacy Customer Portals.

## PushTopic Queries

The PushTopic query is the basis of the PushTopic channel and defines which record create, update, delete, or undelete events generate a notification. This query must be a valid SOQL query. To ensure that notifications are sent in a timely manner, the following requirements apply to PushTopic queries.

- The query `SELECT` clause must include `Id`. For example: `SELECT Id, Name FROM...`
- Only one entity per query.
- The object must be valid for the specified API version.

The fields that you specify in the PushTopic `SELECT` clause make up the body of the notification that is streamed on the PushTopic channel. For example, if your PushTopic query is `SELECT Id, Name, Status__c FROM Invoice_Statement__c`, then the `Id`, `Name` and `Status__c` fields are included in any notifications sent on that channel. Following is an example of a notification message that might appear in that channel:

```
{
  "channel": "/topic/InvoiceStatementUpdates",
  "data": {
    "event": {
      {
        "createdDate": "2016-03-29T16:40:08.208Z",
        "replayId": 13,
        "type": "created"
      }
    }
  }
}
```

```

    },
    "subject":
    {
        "Name": "INV-0001",
        "Id": "a00D000000806y8IAA",
        "Status__c": "Open"
    }
}

```

If you change a PushTopic query, those changes take effect immediately on the server. A client receives events only if they match the new SOQL query. If you change a PushTopic Name, live subscriptions are not affected. New subscriptions must use the new channel name.



**Note:** In API version 37.0 and later, the time format of the `createdDate` field value has changed to make it consistent with the time format used in the Salesforce app. The time portion now ends with a `Z` suffix instead of `+0000`. Both suffixes denote a UTC time zone.

## Security and the PushTopic Query

Subscribers receive notifications about records that were created, updated, deleted, or undeleted if they have:

- Field-level security access to the fields specified in the `WHERE` clause
- Read access on the object in the query
- Read access on the PushTopic
- Visibility of the new or modified record based on sharing rules

If the subscriber doesn't have access to fields referenced in the query `SELECT` clause, then those fields aren't included in the notification. If the subscriber doesn't have access to all fields referenced in the query `WHERE` clause, then they don't receive the notification. In addition to the field permissions that the subscriber must have, the user who creates or manipulates the record, which causes the notification to be generated, must have sufficient permissions to the fields in the `SELECT` clause. Otherwise, the notification that the subscriber receives doesn't include the fields the user doesn't have access to even if the subscriber has access to them.

For example, assume a user tries to subscribe to a PushTopic with this `Query` value.

```

SELECT Id, Name, SSN__c
FROM Employee__c
WHERE Bonus_Received__c = true AND Bonus_Amount__c > 20000

```

If the subscriber doesn't have access to `Bonus_Received__c` or `Bonus_Amount__c`, the subscription fails. If the subscriber doesn't have access to `SSN__c`, then it isn't returned in the notification. If the subscriber has access to the `SSN__c` field but the user who created or manipulated the record doesn't, the subscriber doesn't receive this field in the notification.

If the subscriber already successfully subscribed to the PushTopic, but the field-level security changes and the user no longer has access to one of the fields referenced in the `WHERE` clause, then no streamed notifications are sent.

## Supported PushTopic Queries

All custom objects are supported in PushTopic queries. A subset of standard objects is supported in PushTopic queries, including:

- Account
- Campaign
- Case

- Contact
- ContractLineItem
- Entitlement
- Lead
- LiveChatTranscript
- MessagingSession
- Opportunity
- Quote
- QuoteLineItem
- ServiceAppointment
- ServiceContract
- Task
- UserServicePresence
- WorkOrder
- WorkOrderLineItem

It's possible that not all objects are available in your org. Some objects require specific feature settings and permissions to be enabled.

 **Important:** Tasks that are created or updated using these methods don't appear in task object topics in the streaming API.

- Lead conversion
- Entity merge
- Mass email contacts/leads

Also, the standard SOQL operators and most SOQL statements and expressions are supported. Some SOQL statements aren't supported. See [Unsupported PushTopic Queries](#).

The following are examples of supported SOQL statements.

- Custom object

```
SELECT Id, MyCustomField__c FROM MyCustomObject__c
```

- Standard objects (can include custom fields)

- Account

```
SELECT Id, Name FROM Account WHERE NumberOfEmployees > 1000
```

- Campaign

```
SELECT Id, Name FROM Campaign WHERE Status = 'Planned'
```

- Case

```
SELECT Id, Subject FROM Case WHERE Status = 'Working' AND IsEscalated = TRUE
```

- Contact

```
SELECT Id, Name, Email FROM Contact;
```

- Lead

```
SELECT Id, Company FROM Lead WHERE Industry = 'Computer Services'
```

- Opportunity

```
SELECT Id, Name, Amount FROM Opportunity WHERE CloseDate < 2011-06-14
```

- Task

```
SELECT Id, Subject, IsClosed, Status FROM Task WHERE IsClosed = TRUE
```

## Considerations

System fields—In general, changes in fields that the system populates, such as `LastModifiedDate` and `LastActivityDate`, don't trigger notifications.


Task records—Check out these considerations for task records.

- To receive notifications on the `IsClosed` field, the subscriber must subscribe to the `Status` field referenced in the query.
- To receive notifications on the `WhoCount` and `WhatCount` fields, the subscriber must subscribe to the `WhoId` and `WhatId` fields. Subscriptions based only on the `WhoCount` or `WhatCount` fields aren't supported.
- To receive notifications for task record deletions, the subscribed user must have the View All Data or the Modify All Data permission when the organization-wide sharing setting for activities is private.

Custom picklist values—If the PushTopic query references a custom picklist with a global value set, the PushTopic notification message includes the API name of the value instead of the label. For example, you have a custom picklist field with the name `Colour__c`. One picklist value has the API name of `RD` and the label of `Red`. The PushTopic query references the `Colour__c` picklist field: `SELECT Id, Colour__c, Salutation FROM Contact`. The PushTopic notification message contains the `Colour__c` field with this value: `RD`.

## Compound Fields in PushTopic Queries

By default, the support of compound fields, such as Name or Address fields, depends on which fields are present in the PushTopic query. For Name compound fields, you must specify the Name field. For Address and Geolocation fields, you must specify the constituent fields.

 **Note:** If the PushTopic field `NotifyForFields` is set to `All`, compound fields are supported. In this case, it's not necessary to explicitly reference compound or constituent fields in the PushTopic query. The special behavior listed in the following sections applies only for the default `NotifyForFields` setting (`Referenced`) or when `NotifyForFields` is set to `Select` or `Where`.

### Name Compound Field

To detect changes on the Name compound field, include the Name field in the `SELECT` or `WHERE` clause. The constituent fields, such as `firstName` and `lastName`, are optional, but the Name field is required. The returned notification message includes all constituent field values. If the Name field is omitted, changes can't be detected, even if the constituent fields are present.

The following table shows supported and unsupported `SELECT` statements. These statements contain fields for the Name compound field on Contact or Lead.

Fields	Supported?
<code>SELECT Id, Name</code>	Yes

Fields	Supported?
<code>SELECT Id, Name, firstName, lastName</code>	Yes
<code>SELECT Id, firstName, lastName</code>	No

## Address Compound Field

To detect changes of Address compound fields, include the constituent fields in the SELECT or WHERE clause. The Address field, such as MailingAddress on Contact or ShippingAddress on Account, is optional, but the constituent fields are required. If the constituent fields are omitted, changes can't be detected, even if the Address field is present.

The following table shows supported and unsupported SELECT statements. These statements contain MailingAddress fields of Contact.

Fields	Supported?
<code>SELECT Id, MailingAddress</code>	No
<code>SELECT Id, MailingAddress, MailingCity, MailingStreet</code>	Yes
<code>SELECT Id, MailingCity, MailingStreet</code>	Yes

## Geolocation Compound Field

To detect changes of Geolocation compound fields, include the latitude and longitude constituent fields in the SELECT or WHERE clause. The Geolocation field is optional, but the constituent fields are required. If the constituent fields are omitted, changes can't be detected, even if the Geolocation field is present.

The following table shows supported and unsupported SELECT statements. These statements contain a custom Geolocation field called `location__c` and its constituent fields.

Fields	Supported?
<code>SELECT Id, location__c</code>	No
<code>SELECT Id, location__c, location__latitude__s, location__longitude__s</code>	Yes
<code>SELECT Id, location__latitude__s, location__longitude__s</code>	Yes

## Unsupported PushTopic Queries

These SOQL statements aren't supported in PushTopic queries.

- Queries without an `Id` in the selected fields list
- Semi-joins and anti-joins
  - Example query: `SELECT Id, Name FROM Account WHERE Id IN (SELECT AccountId FROM Contact WHERE Title = 'CEO')`

- Error message: `INVALID_FIELD`, semi/anti join sub-selects are not supported
- Aggregate queries (queries that use AVG, MAX, MIN, and SUM)
  - Example query: `SELECT Id, AVG(AnnualRevenue) FROM Account`
  - Error message: `INVALID_FIELD`, Aggregate queries are not supported
- COUNT
  - Example query: `SELECT Id, Industry, Count(Name) FROM Account`
  - Error message: `INVALID_FIELD`, Aggregate queries are not supported
- LIMIT
  - Example query: `SELECT Id, Name FROM Contact LIMIT 10`
  - Error message: `INVALID_FIELD`, 'LIMIT' is not allowed
- Relationships aren't supported, but you can reference an ID:
  - Example query: `SELECT Id, Contact.Account.Name FROM Contact`
  - Error message: `INVALID_FIELD`, relationships are not supported
- Searching for values in Text Area fields.
- ORDER BY
  - Example query: `SELECT Id, Name FROM Account ORDER BY Name`
  - Error message: `INVALID_FIELD`, 'ORDER BY' clause is not allowed
- GROUP BY
  - Example query: `SELECT Id, AccountId FROM Contact GROUP BY AccountId`
  - Error message: `INVALID_FIELD`, 'Aggregate queries are not supported'
- Formula fields in WHERE clauses (But formula fields are supported in SELECT clauses.)
- NOT
  - Example query: `SELECT Id FROM Account WHERE NOT Name = 'Salesforce.com'`
  - Error message: `INVALID_FIELD`, 'NOT' is not supported

To make this a valid query, change it to `SELECT Id FROM Account WHERE Name != 'Salesforce.com'`.



**Note:** The NOT IN phrase is supported in PushTopic queries.

- OFFSET
  - Example query: `SELECT Id, Name FROM Account WHERE City = 'New York' OFFSET 10`
  - Error message: `INVALID_FIELD`, 'OFFSET' clause is not allowed
- TYPEOF
  - Example query: `SELECT TYPEOF Owner WHEN User THEN LastName ELSE Name END FROM Case`
  - Error message: `INVALID_FIELD`, 'TYPEOF' clause is not allowed
- Roll-up summary fields. If a PushTopic query contains a roll-up summary field for an object, notifications aren't be sent for changes (create, update, delete, and undelete) in that field.

## Event Notification Rules

Notifications are generated for record events based on how you configure your PushTopic. The Streaming API matching logic uses the `NotifyForOperationCreate`, `NotifyForOperationUpdate`, `NotifyForOperationDelete`, `NotifyForOperationUndelete`, and `NotifyForFields` fields in a PushTopic record to determine whether to generate a notification.

Clients must connect using the `cometd/29.0` (or later) Streaming API endpoint to receive delete and undelete event notifications.

## Events

Events that may generate a notification are the creation, update, delete, or undelete of a record. The PushTopic `NotifyForOperationCreate`, `NotifyForOperationUpdate`, `NotifyForOperationDelete`, and `NotifyForOperationUndelete` fields enable you to specify which events may generate a notification in that PushTopic channel. The fields are set as follows:

Field	Description
<code>NotifyForOperationCreate</code>	<code>true</code> if a create operation should generate a notification, otherwise, <code>false</code> .
<code>NotifyForOperationDelete</code>	<code>true</code> if a delete operation should generate a notification, otherwise, <code>false</code> .
<code>NotifyForOperationUndelete</code>	<code>true</code> if an undelete operation should generate a notification, otherwise, <code>false</code> .
<code>NotifyForOperationUpdate</code>	<code>true</code> if an update operation should generate a notification, otherwise, <code>false</code> .

In API version 28.0 and earlier, you use the `NotifyForOperations` field to specify which events generate a notification, and can only specify create or update events. The `NotifyForOperations` values are:

<code>NotifyForOperations</code> Value	Description
All (default)	Evaluate a record to possibly generate a notification whether the record has been created or updated.
Create	Evaluate a record to possibly generate a notification only if the record has been created.
Update	Evaluate a record to possibly generate a notification only if the record has been updated.
Extended	A value of <code>Extended</code> means that neither create or update operations are set to generate events. This value is provided to allow clients written to API version 28.0 or earlier to work with Salesforce organizations configured to generate delete and undelete notifications.

The event field values together with the `NotifyForFields` value provides flexibility when configuring when you want to generate notifications using Streaming API.

## Notifications

After a record is created or updated (an event), the record is evaluated against the PushTopic query and a notification might be generated. A notification is the message sent to the channel as the result of an event. The notification is a JSON formatted message. The PushTopic field `NotifyForFields` specifies how the record is evaluated against the PushTopic query. The `NotifyForFields` values are:

<code>NotifyForFields</code> Value	Description
All	Notifications are generated for all record field changes, provided the evaluated records match the criteria specified in the WHERE clause.
Referenced (default)	Changes to fields referenced in the SELECT and WHERE clauses are evaluated. Notifications are generated for the evaluated records only if they match the criteria specified in the WHERE clause.
Select	Changes to fields referenced in the SELECT clause are evaluated. Notifications are generated for the evaluated records only if they match the criteria specified in the WHERE clause.
Where	Changes to fields referenced in the WHERE clause are evaluated. Notifications are generated for the evaluated records only if they match the criteria specified in the WHERE clause.

The fields that you specify in the PushTopic query SELECT clause are contained in the notification message.

### NotifyForFields Set to All


When you set the value of `PushTopic.NotifyForFields` to `All`, a change to any field value in the record causes the Streaming API matching logic to evaluate the record to determine whether to generate a notification. Changes to record field values cause this evaluation whether or not those fields are referenced in the PushTopic query SELECT clause or WHERE clause.

Event	A notification is generated when
Record is created	The record field values match the values specified in the WHERE clause.
Record is updated	The record field values match the values specified in the WHERE clause.

### Examples

PushTopic Query	Result
SELECT Id, f1, f2, f3 FROM Invoice_Statement__c	Generates a notification if any field values in the record have changed.
SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'	Generates a notification if any field values in the record have changed and f3 and f4 match the values in the WHERE clause.
SELECT Id FROM Invoice_Statement__c	When Id is the only field in the SELECT clause, a notification is generated if any field values have changed.

PushTopic Query	Result
<pre>SELECT Id FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'</pre>	Generates a notification if any field values in the record have changed and f3 and f4 match the values in the WHERE clause.
<pre>SELECT Id FROM Invoice_Statement__c WHERE Id IN ('a07B00000000KWZ7IAO', 'e10R00000000KEU9IAO', 'v32B00000000KWZ7YEP')</pre>	Generates a notification if any field values in the record have changed and the record ID is contained in the WHERE clause IN list.
<pre>SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE Id IN ('a07B00000000KWZ7IAO', 'e10R00000000KEU9IAO', 'v32B00000000KWZ7YEP')</pre>	Generates a notification if any field values in the record have changed and the record ID is contained in the WHERE clause IN list.
<pre>SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz' AND Id IN ('a07B00000000KWZ7IAO', 'e10R00000000KEU9IAO', 'v32B00000000KWZ7YEP')</pre>	Generates a notification if any field values in the record have changed, f3 and f4 match the WHERE clause, and the record ID is contained in the WHERE clause IN list.

 **Warning:** Use caution when setting `NotifyForFields` to `All`. When you use this value, then notifications are generated for all record field changes as long as the new field values match the values in the WHERE clause. Therefore, the number of generated notifications could potentially be large, and you may hit the daily quota of events allocation. In addition, because every record change is evaluated and many notifications may be generated, this causes a heavier load on the system.

### NotifyForFields Set to Referenced

When you set the value of `PushTopic.NotifyForFields` to `Referenced`, a change to any field value in the record as long as that field is referenced in the query `SELECT` clause or `WHERE` clause causes the Streaming API matching logic to evaluate the record to determine whether to generate a notification.

If the `PushTopic.NotifyForFields` value is `Referenced`, the `PushTopic` query must have a `SELECT` clause with at least one field other than `ID` or a `WHERE` clause with at least one field other than `Id`.

Event	A notification is generated when
Record is created	The record field values match the values specified in the WHERE clause.
Record is updated	The following conditions are met.

Event	A notification is generated when
	<ul style="list-style-type: none"> <li>A change occurs in one or more record fields that are specified in the PushTopic query SELECT or WHERE clause.</li> <li>For fields specified in the WHERE clause, all record field values match the values in the WHERE clause.</li> </ul>

## Examples

PushTopic Query	Result
SELECT Id, f1, f2, f3 FROM Invoice_Statement__c	Generates a notification if f1, f2, or f3 have changed.
SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'	Generates a notification if f1, f2, f3, or f4 have changed and f3 and f4 match the values in the WHERE clause.
SELECT Id FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'	Generates a notification if f3 and f4 have changed and f3 and f4 match the values in the WHERE clause.
SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE Id IN ('a07B00000000KWZ7IAO', 'e10R00000000KEU9IAO', 'v32B00000000KWZ7YEP')	Generates a notification if f1 or f2 have changed and the record ID is contained in the WHERE clause IN list.
SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz' AND Id IN ('a07B00000000KWZ7IAO', 'e10R00000000KEU9IAO', 'v32B00000000KWZ7YEP')	Generates a notification if f1, f2, f3, or f4 have changed, f3 and f4 match the values in the WHERE clause, and the ID is contained in the WHERE clause IN list.

## NotifyForFields Set to Select

When you set the value of `PushTopic.NotifyForFields` to `Select`, a change to any field value in the record, as long as that field is referenced in the query SELECT clause, causes the Streaming API matching logic to evaluate the record to determine whether to generate a notification.

If the `PushTopic.NotifyForFields` value is `Select`, the PushTopic query must have a SELECT clause with at least one field other than ID.

Event	A notification is generated when
Record is created	The record field values match the values specified in the WHERE clause.
Record is updated	<p>The following conditions are met.</p> <ul style="list-style-type: none"> <li>A change occurs in one or more record fields specified in the PushTopic query SELECT clause.</li> </ul>

Event	A notification is generated when
	<ul style="list-style-type: none"> <li>The record values of the fields specified in the WHERE clause all match the values in the PushTopic query WHERE clause.</li> </ul>

## Examples

PushTopic Query	Result
SELECT Id, f1, f2, f3 FROM Invoice_Statement__c	Generates a notification if f1, f2, or f3 have changed.
SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'	Generates a notification if f1 or f2 have changed and f3 and f4 match the values in the WHERE clause.
SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE Id IN ('a07B00000000KWZ7IAO', 'e10R00000000KEU9IAO', 'v32B00000000KWZ7YEP')	Generates a notification if f1 or f2 have changed and ID is contained in the WHERE clause IN list.
SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz' AND Id IN ('a07B00000000KWZ7IAO', 'e10R00000000KEU9IAO', 'v32B00000000KWZ7YEP')	Generates a notification if f1 or f2 have changed, f3 and f4 match the values in the WHERE clause, and the ID is contained in the WHERE clause IN list.

## NotifyForFields Set to Where

When you set the value of `PushTopic.NotifyForFields` to `Where`, a change to any field value in the record as long as that field is referenced in the query WHERE clause causes the Streaming API matching logic to evaluate the record to determine whether to generate a notification.

If the `PushTopic.NotifyForFields` value is `Where`, the PushTopic query must have a WHERE clause with at least one field other than `Id`.

Event	A notification is generated when
Record is created	The record field values match the values specified in the WHERE clause.
Record is updated	<p>The following conditions are met.</p> <ul style="list-style-type: none"> <li>A change occurs in one or more record fields specified in the PushTopic query WHERE clause.</li> <li>The record values of the fields specified in the WHERE clause all match the values in the PushTopic query WHERE clause.</li> </ul>

## Examples

PushTopic Query	Result
SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'	Generates a notification if f3 or f4 have changed and the values match the values in the WHERE clause.
SELECT Id FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'	Generates a notification if f3 or f4 have changed and the values match the values in the WHERE clause.
SELECT Id, f1, f2 FROM Invoice_Statement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz' AND Id IN ('a07B0000000KWZ7IAO', 'e10R0000000KEU9IAO', 'v32B0000000KWZ7YEP')	Generates a notification if f3 or f4 have changed, f3 and f4 match the values in the WHERE clause, and the record ID is contained in the WHERE clause IN list.

## Notification Scenarios

Following is a list of example scenarios and the field values you need in a PushTopic record to generate notifications.

Scenario	Configuration
You want to receive all notifications of all record updates.	<ul style="list-style-type: none"> <li>• <b>MyPushTopic</b>.Query = SELECT Id, Name, Description__c FROM Invoice_Statement__c</li> <li>• <b>MyPushTopic</b>.NotifyForFields = All</li> </ul>
You want to receive notifications of all record changes only when the Name or Amount fields change. For example, if you're maintaining a list view.	<ul style="list-style-type: none"> <li>• <b>MyPushTopic</b>.Query = SELECT Id, Name, Amount__c FROM Invoice_Statement__c</li> <li>• <b>MyPushTopic</b>.NotifyForFields = Referenced</li> </ul>
You want to receive notification of all record changes made to a specific record.	<ul style="list-style-type: none"> <li>• <b>MyPushTopic</b>.Query = SELECT Id, Name, Amount__c FROM Invoice_Statement__c WHERE Id='a07B0000000KWZ7IAO'</li> <li>• <b>MyPushTopic</b>.NotifyForFields = All</li> </ul>
You want to receive notification only when the Name or Amount field changes for a specific record. For example, if the user is on a detail page and only those two fields are displayed.	<ul style="list-style-type: none"> <li>• <b>MyPushTopic</b>.Query = SELECT Id, Name, Amount__c FROM Invoice_Statement__c WHERE Id='a07B0000000KWZ7IAO'</li> <li>• <b>MyPushTopic</b>.NotifyForFields = Referenced</li> </ul>
You want to receive notification for all invoice statement record changes for vendors in a particular state.	<ul style="list-style-type: none"> <li>• <b>MyPushTopic</b>.Query = SELECT Id, Name, Amount__c FROM Invoice_Statement__c WHERE BillingState__c = 'NY'</li> <li>• <b>MyPushTopic</b>.NotifyForFields = All</li> </ul>

**Scenario**

You want to receive notification for all invoice statement record changes where the invoice amount is \$1,000 or more.

**Configuration**

- ***MyPushTopic***.Query = SELECT Id, Name FROM Invoice\_Statement\_\_c WHERE Amount\_\_c > 999
- ***MyPushTopic***.NotifyForFields = All

## Replay PushTopic Streaming Events

Salesforce stores PushTopic-based events for 24 hours and allows you to retrieve stored and new events. Subscribers can choose which events to receive by using replay options.

For more information about durable events, see [Message Durability](#).

## Code Samples

- [GitHub: Durable PushTopic Streaming Demo](#)
- [GitHub: Streaming Replay Client Extensions](#)

## Filtered Subscriptions

Reduce the number of PushTopic event notifications by specifying record fields to filter on when you subscribe to a channel.

Specify the filter criteria in an expression you append to the subscription URI, as follows.

```
/topic/TopicName?<expression>
```

*TopicName* is the PushTopic name, and *<expression>* is the expression containing one or more conditions. Join conditions with the & operator. Only the & operator is supported. Use this syntax for the *<expression>*.

```
?fieldA=valueA&fieldB=valueB&...
```

Include each field used in a filter condition in the PushTopic query. The & operator acts like the logical OR operator, so record events are matched if any condition is true.



**Example:** This subscription returns event notifications for records whose industry is Energy *or* shipping city is San Francisco.

```
/topic/MyTopic?Industry='Energy'&ShippingCity='San Francisco'
```

The PushTopic query for this subscription includes the `Industry` and `ShippingCity` fields.



**Note:**

- If you use an ID in filter criteria, use the 18-character ID format; 15-character IDs aren't supported.
- When using a shared CometD session with the `empApi` Lightning component, all subscriptions to the same PushTopic must specify the same filter. If the initial subscription doesn't use a filter, the subsequent subscriptions can't. For example, if you subscribe to `/topic/MyTopic?Name='SomeName'`, you can't resubscribe to `/topic/MyTopic` or `/topic/MyTopic?Industry='Energy'` using the same CometD session. Shared CometD sessions apply to subscriptions you make with the `empApi` Lightning component only. The `empApi` component shares the CometD session when you open a second instance of `empApi` in a new tab, or in a new browser for the same Salesforce user session.

## Bulk Subscriptions

You can subscribe to multiple topics at the same time.

To do so, send a JSON array of subscribe messages instead of a single subscribe message. For example this code subscribes to three topics:

```
[
  {
    "channel": "/meta/subscribe",
    "clientId": "Un1q31d3nt1f13r",
    "subscription": "/topic/foo"
  },
  {
    "channel": "/meta/subscribe",
    "clientId": "Un1q31d3nt1f13r",
    "subscription": "/topic/bar"
  },
  {
    "channel": "/meta/subscribe",
    "clientId": "Un1q31d3nt1f13r",
    "subscription": "/topic/baz"
  }
]
```

For more information, see the [Bayeux Specification](#).

## Deactivating a Push Topic

You can temporarily deactivate a PushTopic, rather than deleting it, by setting the `isActive` field to false.

- To deactivate a PushTopic by Id, execute the following Apex code:

```
PushTopic pt = new PushTopic(Id='0IFD0000000008jOAA', IsActive = false);
update(pt);
```

## PushTopic Considerations

---

Keep in mind these considerations when subscribing to PushTopic events.

### IN THIS SECTION:

[PushTopic Notification Message Order](#)

[Considerations for Multiple Notifications in the Same Transaction](#)

## PushTopic Notification Message Order

### PushTopic Notification Message Order with API Version 37.0 and Later

In API version 37.0 and later, Salesforce stores PushTopic notification messages temporarily in the event bus. Before a PushTopic notification message is stored, Salesforce assigns it a replay ID value. Subscribers receive notification messages from the event bus in the order of the replay ID. In general, PushTopic message notification order is the same as the transactions corresponding to the record changes that are committed in Salesforce. One exception is when a record triggers multiple notifications within the same transaction, the last notification is delivered first. For more information, see [Streaming API Notifications Sent in Reverse Order Within a Transaction](#).

This example shows two PushTopic notification messages for two new invoice statements. INV-0001 is created before INV-0002 and its `replayId` value is lower.

```
{
  "data": {
    "event": {
      "createdDate": "2021-08-05T17:49:08.990Z",
      "replayId": 2,
      "type": "created"
    },
    "subject": {
      "Description__c": "New invoice statement #2",
      "Id": "a02RM00000013VrYAI",
      "Status__c": "Open",
      "Name": "INV-0002"
    }
  },
  "channel": "/topic/InvoiceStatementUpdates"
}

{
  "data": {
    "event": {
      "createdDate": "2021-08-05T17:33:48.324Z",
      "replayId": 1,
      "type": "created"
    },
    "subject": {
      "Description__c": "New invoice statement",
      "Id": "a02RM00000013VmYAI",
      "Status__c": "Open",
      "Name": "INV-0001"
    }
  },
}
```

```
"channel": "/topic/InvoiceStatementUpdates"
}
```

## PushTopic Notification Message Order with API Version 36.0 and Earlier

In API version 36.0 and earlier, changes to data in your organization happen sequentially. But the order of the PushTopic event notification messages that you receive isn't guaranteed. On the client side, you can use `createdDate` to order the notification messages returned in a channel. The value of `createdDate` is a UTC date/time value that indicates when the event occurred.

This example shows two PushTopic notification messages for two new invoice statements. INV-0001 is created before INV-0002 and its `createdDate` value is lower than the one for INV-0002.

```
{
  "data": {
    "event": {
      "createdDate": "2013-05-10T18:16:19.000+0000",
      "type": "created"
    },
    "subject": {
      "Description__c": "New invoice statement #2",
      "Id": "a00D00000008pvxcIAA",
      "Status__c": "Open",
      "Name": "INV-0002"
    }
  },
  "channel": "/topic/InvoiceStatementUpdates"
}

{
  "data": {
    "event": {
      "createdDate": "2013-05-10T18:15:11.000+0000",
      "type": "created"
    },
    "subject": {
      "Description__c": "New invoice statement #1",
      "Id": "a00D00000008pvzdIAA",
      "Status__c": "Open",
      "Name": "INV-0001"
    }
  },
  "channel": "/topic/InvoiceStatementUpdates"
}
```

## Considerations for Multiple Notifications in the Same Transaction

Learn about the behavior of Streaming API when multiple notifications are delivered within the same transaction.

## IN THIS SECTION:

[Streaming API Notifications Sent in Reverse Order Within a Transaction](#)

In general, event notifications are delivered in the order of record changes. One exception is that when a record triggers multiple notifications within the same transaction, the last notifications are delivered first.

[Multiple Streaming API Notifications for the Same Record](#)

In API version 37.0 and later, if a field change triggers other field changes on the same record, more notifications are sent for those fields even if they aren't tracked. Also, if a field changes multiple times in a record during a transaction, multiple notifications are sometimes sent. The notifications are sent regardless of whether the field values match the PushTopic query.

[Only Last PushTopic Notification Sent for the Same Record](#)

In API version 36.0 and earlier, when multiple PushTopic notifications are generated for the same record within about one millisecond and in the same transaction, only the last notification is sent.

## Streaming API Notifications Sent in Reverse Order Within a Transaction

In general, event notifications are delivered in the order of record changes. One exception is that when a record triggers multiple notifications within the same transaction, the last notifications are delivered first.

For example, let's say you have a PushTopic for insertions and updates of contact records, and the PushTopic query selects `fieldA`. If a contact is inserted and then an Apex trigger or workflow updates `fieldA` in the same transaction, the order of notifications sent is:

- Notification for the update of `fieldA`
- Notification for the insertion of the record

In this case, the order of notifications depends on the order in which the Lightning Platform commits transactions.

## Multiple Streaming API Notifications for the Same Record

In API version 37.0 and later, if a field change triggers other field changes on the same record, more notifications are sent for those fields even if they aren't tracked. Also, if a field changes multiple times in a record during a transaction, multiple notifications are sometimes sent. The notifications are sent regardless of whether the field values match the PushTopic query.

Multiple notifications are sent because the first change causes the record to be reevaluated for more changes.


For example, let's say you have a PushTopic that tracks fields specified in the WHERE clause. This snippet inserts the PushTopic with a query on the Case object.

```
PushTopic pushTopic = new PushTopic();
pushTopic.Name = 'MyPushTopicExample';
pushTopic.Query =
    'SELECT Id, Status, CaseNumber, OwnerId, Priority FROM Case WHERE Priority = \'High\'';
pushTopic.ApiVersion = 37.0;
pushTopic.NotifyForFields = 'Where';
insert pushTopic;
```

Suppose that the Case object has a custom field called `MyField__c`, which the PushTopic query doesn't track in the WHERE clause. A notification is generated for a record when the tracked `Priority` field value changes to `High`. When a trigger or workflow later updates another field, such as `MyField__c`, a notification is generated even though this field isn't tracked. The extra notification is sent because the record met the PushTopic criteria the first time. And because it's still the same transaction, the record remains the source of the notifications.

Another scenario is when a field's value changes in the same transaction. For example, a case record is created with `Priority` set to `Medium`, which doesn't match the PushTopic query. Next, a trigger or workflow changes the `Priority` to `High` in the same

transaction, which matches the PushTopic query. As a result, two notifications are sent, one for each change, even though the first change doesn't match the criteria. Also, the `Priority` value in both notifications contains the latest value: `High`.

 **Note:** In API version 36.0 and earlier, only a single notification is sent for the last record change, and there are no additional notifications. For more information, see [Only Last PushTopic Notification Sent for the Same Record](#).

## Only Last PushTopic Notification Sent for the Same Record


In API version 36.0 and earlier, when multiple PushTopic notifications are generated for the same record within about one millisecond and in the same transaction, only the last notification is sent.

The other notifications are suppressed because notifications are tracked at the millisecond level. When multiple notifications happen within a transaction at the same time—less than one millisecond—only the last notification can be delivered.

For example, let's say you have a PushTopic for insertions and updates of contact records, and the PushTopic query selects `fieldA`. If a contact is inserted and then an Apex trigger or workflow updates `fieldA` within a short time, only the notification for the update is sent. A notification isn't sent for the contact creation.

However, if the elapsed time between the notifications is over one millisecond, all notifications are sent for the same record, and no notification is suppressed.


To learn more about transactions, see [Apex Transactions](#) in the Apex Developer Guide and [Flows in Transactions](#) in the Salesforce Help.

 **Note:** In API version 37.0 and later, notifications are tracked by a unique ID and don't depend on the time when they were generated. All notifications for the same record within one transaction are sent, and notifications aren't suppressed.

## PushTopic Streaming Allocations

These default allocations apply to consumers of PushTopic Streaming in all API versions.

Description	Performance and Unlimited Editions	Enterprise Edition	All other supported editions
Maximum number of topics (PushTopic records) per org	100	50	40
Maximum number of concurrent clients (subscribers) across all channels and for all event types	2,000	1,000	20
Maximum number of delivered event notifications within a 24-hour period, shared by all CometD clients	1,000,000	200,000	50,000 (10,000 for free orgs)
Socket timeout during connection (CometD session)	110 seconds	110 seconds	110 seconds
Timeout to reconnect after successful connection (keepalive)	40 seconds	40 seconds	40 seconds
Maximum length of the SOQL query in the <code>Query</code> field of a PushTopic record	1,300 characters	1,300 characters	1,300 characters
Maximum length for a PushTopic name	25 characters	25 characters	25 characters

 **Note:** For free orgs, the maximum number of events within a 24-hour period is 10,000. Free orgs include Developer Edition orgs and trial orgs (all editions), such as partner test and demo orgs created through the Environment Hub. Sandboxes get the same allocations as their associated production orgs.

If you exceed the default event delivery allocation, you receive this error: `403::Organization total events daily limit exceeded`. The error is returned in the Bayeux `/meta/connect` channel when a CometD subscriber first connects or in an existing subscriber connection. For more information, see [Streaming API Error Codes](#). PushTopic event notifications that are generated after exceeding the allocation are stored in the event bus. You can retrieve stored event messages as long as they are within the retention window of 24 hours.

If you have scenarios that warrant an increase in the number of delivered event notifications within a 24-hour period, contact Salesforce.

SEE ALSO:

[Monitoring Event Usage](#)

## Reference: PushTopic

---


See [PushTopic](#) in the *Object Reference for Salesforce and Lightning Platform*.

## CHAPTER 5 Generic Events (Legacy)

### In this chapter ...

- [Replay Generic Streaming Events with Durable Generic Streaming](#)
- [Generic Streaming Quick Start](#)
- [Generic Streaming Allocations](#)
- [Reference: StreamingChannel](#)
- [Reference: Streaming Channel Push REST API](#)

Use generic events to send custom notifications that aren't tied to Salesforce data changes.


 **Important:** Generic Events is a legacy product. Salesforce no longer enhances Generic Events with new features and provides limited support for this product. Instead of Generic Events, consider using Platform Events. To learn about Platform Events, see the [Platform Events Developer Guide](#) and the [Platform Events Basics Trailhead module](#).

Use generic streaming when you want to send and receive custom notifications. Use generic streaming to:

- Publish and receive arbitrary payloads in JSON without a predefined event schema
- Broadcast notifications to a target set of users, specific teams, or your entire org
- Send notifications for events that are external to Salesforce

To use generic events, you need:

- A [StreamingChannel](#) that defines the channel, with a name that is case-sensitive
- One or more clients subscribed to the channel
- The [Streaming Channel Push](#) REST API resource to monitor and invoke push events on the channel

 **Note:** StreamingChannel isn't available for users in Experience Cloud sites, including Partner Community and Customer Community users. Also, StreamingChannel isn't available in legacy Customer Portals.

## Replay Generic Streaming Events with Durable Generic Streaming

---

A client can receive generic streaming events after it subscribes to a channel and as long as the Salesforce session is active. Events sent before a client subscribes to a channel or after a subscribed client disconnects from the Salesforce session are missed. However, a client can fetch the missed events within the 24-hour retention window by using Durable Generic Streaming.

For more information about durable events, see [Message Durability](#).

### Code Sample

See these code samples about replaying generic streaming events.

- [Generic Streaming Quick Start](#)
- [Example: Subscribe to and Replay Events Using a Visualforce Page](#)

## Generic Streaming Quick Start

---

This quick start shows you how to get started with generic streaming in Streaming API. This quick start takes you step-by-step through the process of using Streaming API to receive a notification when an event is pushed via REST and lets you specify replay options.

### IN THIS SECTION:

#### [Create a Streaming Channel](#)

Create a new StreamingChannel object by using the Salesforce UI.

#### [Run a Java Client with Username and Password Login](#)

Run a Java client that uses EMP Connector to subscribe to the channel with username and password authentication.

#### [Run a Java Client with OAuth Bearer Token Login](#)

Run a Java client that uses EMP Connector to subscribe to the channel with OAuth authentication.

#### [Generate Events Using REST](#)

Use the Streaming Channel Push REST API resource to generate event notifications to channel subscribers.

## Create a Streaming Channel

Create a new StreamingChannel object by using the Salesforce UI.

You must have the proper Streaming API permissions enabled in your organization.

1. Log in to your Developer Edition organization.
2. If you're using Salesforce Classic, under All Tabs (+), select **Streaming Channels**. If you're using Lightning Experience, from the App Launcher, select **All Items**, and then click **Streaming Channels**.
3. On the Streaming Channels page, click **New** to create a streaming channel.
4. Enter `/u/notifications/ExampleUserChannel` in **Streaming Channel Name**, and an optional description. Your new Streaming Channel page should look something like this:

5. Select **Save**. You've just created a streaming channel that clients can subscribe to for notifications.

StreamingChannel is a regular, creatable Salesforce object, so you can also create one programmatically using Apex or any data API like SOAP API or REST API.

Also, if you need to restrict which users can receive or send event notifications, you can use user sharing on the StreamingChannel to control this. Channels shared with public read-only or read-write access send events only to clients subscribed to the channel that also are using a user session associated with the set of shared users or groups. Only users with read-write access to a shared channel can generate events on the channel, or modify the actual StreamingChannel record. To modify user sharing for a StreamingChannel, from Setup, enter *Sharing Settings* in the Quick Find box, then select **Sharing Settings** and create or modify a StreamingChannel sharing rule.

Generic streaming also supports dynamic streaming channel creation, which creates a StreamingChannel when a client first subscribes to the channel. To enable dynamic streaming channels in your org, from Setup, enter *User Interface* in the Quick Find box, then select **User Interface**. Enable **Enable Dynamic Streaming Channel Creation**. You can also enable dynamic channel creation in Metadata API using EventSettings.

## Run a Java Client with Username and Password Login

Run a Java client that uses EMP Connector to subscribe to the channel with username and password authentication.

1. Get the EMP Connector project from GitHub. See [Download and Build the Project](#).
2. In the `/src/main/java/com/salesforce/emp/connector/example` folder, open the `LoginExample.java` source file.
3. Run the `LoginExample` class and provide arguments.

Argument	Value
<code>username</code>	Username of the logged-in user.
<code>password</code>	Password for the <code>username</code> (or logged-in user).
<code>channel</code>	<code>/u/notifications/ExampleUserChannel</code>

Optionally, to receive different events, you can include a replay ID as the last argument. Valid values are:

- `-1` — Get all new events sent after subscription. This option is the default.
- `-2` — Get all new events sent after subscription and all past events within the retention window. Use `-2` sparingly. If a large volume of event messages is stored, retrieving all event messages can slow performance.
- Specific number — Get all events that occurred after the event with the specified replay ID.

4. When you run this client app and generate notifications using the REST resource, the output will look something like:

```
Subscribed: Subscription [/u/notifications/ExampleUserChannel:-1]
Received:
{payload=Broadcast message to all subscribers,
event={createdDate=2016-12-13T00:57:36.020Z, replayId=1}}
Received:
{payload=Another message, event={createdDate=2016-12-13T00:58:16.591Z, replayId=2}}
```

Generally, don't handle usernames and passwords of others when running code in production. In a production environment, delegate the login to OAuth. The next step connects to Streaming API with OAuth.

## Run a Java Client with OAuth Bearer Token Login

Run a Java client that uses EMP Connector to subscribe to the channel with OAuth authentication.

Obtain an OAuth bearer access token for your Salesforce user. You supply this access token in the connector example.

See [Set Up Authorization with OAuth 2.0](#). Also see [Authorize Apps with OAuth](#).

Let's run an example that uses OAuth bearer token login.

1. Get the EMP Connector project from GitHub. See [Download and Build the Project](#).
2. In the `/src/main/java/com/salesforce/emp/connector/example` folder, open the `BearerTokenExample.java` Java source file.
3. Run the `BearerTokenExample` class, and provide the following argument values.

Argument	Value
<code>username</code>	Username of the logged-in user.
<code>password</code>	Password for the <code>username</code> (or logged-in user).
<code>channel</code>	<code>/u/notifications/ExampleUserChannel</code>

Optionally, to receive different events, you can include a replay ID as the last argument. Valid values are:

- `-1` — Get all new events sent after subscription. This option is the default.
- `-2` — Get all new events sent after subscription and all past events within the retention window. Use `-2` sparingly. If a large volume of event messages is stored, retrieving all event messages can slow performance.
- Specific number — Get all events that occurred after the event with the specified replay ID.

4. When you run this client app and generate notifications using the REST resource, the output will look something like:

```
Subscribed: Subscription [/u/notifications/ExampleUserChannel:-1]
Received:
{payload=Broadcast message to all subscribers,
event={createdDate=2016-12-13T00:57:36.020Z, replayId=1}}
Received:
{payload=Another message, event={createdDate=2016-12-13T00:58:16.591Z, replayId=2}}
```

In the next step, you learn how to generate notifications using REST.

## Generate Events Using REST

Use the Streaming Channel Push REST API resource to generate event notifications to channel subscribers.

Prerequisites:

You use Postman to perform these steps. Before you begin, set up Postman by completing [Quick Start: Connect Postman to Salesforce](#) in *Trailhead*.

1. After setting up Postman and authenticating, get the streaming channel ID by performing a SOQL query using Tooling API.

- a. In Postman, expand **REST**, and click **Query**.
- b. Click **Params**, then for the `q` value, enter this query.

```
SELECT Name, ID FROM StreamingChannel
```

- c. Click **Send**.
- d. The response contains the StreamingChannel ID for `/u/notifications/ExampleUserChannel`. Note this ID for the next step.

2. Publish an event message using the `push` REST API resource.

- a. In Postman, expand **REST** > **Object**, and click **Object Create**.
- b. Select **POST**, and append this path to the request URL after replacing `<Streaming Channel ID>` with the ID you copied earlier: `/<Streaming Channel ID>/push`.

The full URL becomes:

```
{{_endpoint}}/services/data/v{{version}}/objects/:SOBJECT_API_NAME/<Streaming Channel ID>/push
```

- c. Click **Params**, then under **Path Variables**, enter `StreamingChannel` for `SOBJECT_API_NAME`.
- d. Click **Body**, and enter this body value.

```
{
  "pushEvents": [
    {
      "payload": "Broadcast message to all subscribers",
      "userIds": []
    }
  ]
}
```

- e. Click **Send**.

The request sends the event to all subscribers on the channel. You receive the notification with the payload text in your Java client. The REST method response indicates the number of subscribers the event was sent to (in this case, `-1`, because the event was set to broadcast to all subscribers).

This example shows the notification message received.

```
{
  "clientId": "a1ps4wpe52qytvcvbsko09tapc",
  "data": {
    "event": {
      "createdDate": "2016-03-29T19:05:28.334Z",
```

```

    "replayId":55
  },
  "payload":"Broadcast message to all subscribers"
},
"channel":"/u/notifications/ExampleUserChannel"
}

```

You've successfully sent a notification to a subscriber using generic streaming. You can specify the list of subscribed users to send notifications to instead of broadcasting to all subscribers. Also, you can use the GET method of the Streaming Channel Push REST API resource to get a list of active subscribers to the channel.

## Generic Streaming Allocations

These default allocations apply to consumers of generic streaming.

Description	Performance and Unlimited Editions	Enterprise Edition	Professional Edition	Free Orgs
Maximum streaming channels per org	1,000	1,000	1,000	200
Maximum number of concurrent clients (subscribers) across all channels and for all event types	2,000	1,000	20	20
Maximum number of delivered event notifications within a 24-hour period, shared by all CometD clients, with Durable Generic Streaming (API version 37.0 and later)	1,000,000	200,000	100,000	10,000
Maximum number of delivered event notifications within a 24-hour period, shared by all CometD clients, with generic streaming (API version 36.0 and earlier)	100,000	100,000	100,000	10,000



**Note:** Free orgs include Developer Edition orgs and trial orgs (all editions), such as partner test and demo orgs created through the Environment Hub. Sandboxes get the same allocations as their associated production orgs.

If you exceed the default event delivery allocation, you receive this error: `403::Organization total events daily limit exceeded`. The error is returned in the Bayeux `/meta/connect` channel when a CometD subscriber first connects or in an existing subscriber connection. For more information, see [Streaming API Error Codes](#). Generic event notifications that are generated after exceeding the allocation are stored in the event bus. You can retrieve stored event messages as long as they are within the retention window of 24 hours.

If you have scenarios that warrant an increase in the number of delivered event notifications within a 24-hour period, contact Salesforce.

SEE ALSO:

[Monitor PushTopic and Generic Event Usage with the REST API](#)

## Reference: StreamingChannel

See [StreamingChannel](#) in the *Object Reference for Salesforce and Lightning Platform*.

## Reference: Streaming Channel Push REST API

---

Gets subscriber information, and pushes notifications for streaming channels.

### Syntax

#### URI

/vXX.X/subjects/StreamingChannel/**Channel ID**/push

#### Available since release

29.0

#### Formats

JSON, XML

#### HTTP methods

GET, POST

#### Authentication

Authorization: Bearer *token*

#### Request body

For GET, no request body required. For POST, a request body that provides the push notification payload. This contains the following fields:

Name	Type	Description
pushEvents	array of push event payloads	List of event payloads to send notifications for.

Each push event payload contains the following fields:

Name	Type	Description
payload	string	Information sent with notification. Cannot exceed 3,000 single-byte characters.
userIds	array of User IDs	List of subscribed users to send the notification to. If this array is empty, the notification will be broadcast to all subscribers on the channel.

#### Request parameters

None

#### Response data

For GET, information on the channel and subscribers is returned in the following fields:

Name	Type	Description
OnlineUserIds	array of User IDs	User IDs of currently subscribed users to this channel.
ChannelName	string	Name of the channel, for example, <i>/u/notifications/ExampleUserChannel</i> .

For POST, information on the channel and payload notification results is returned in an array of push results, each of which contains the following fields:

Name	Type	Description
<code>fanoutCount</code>	number	The number of subscribers that the event got sent to. This is the count of subscribers specified in the POST request that were online. If the request was broadcast to all subscribers, <code>fanoutCount</code> will be <code>-1</code> . If no active subscribers were found for the channel, <code>fanoutCount</code> will be <code>0</code> .
<code>userOnlineStatus</code>	array of User online status information	List of User IDs the notification was sent to and their listener status. If <code>true</code> the User ID is actively subscribed and listening, otherwise <code>false</code> .

## Example

The following is an example JSON response of a GET request for `services/data/v29.0/subjects/StreamingChannel/0M6D00000000g7KXA/push`:

```
{
  "OnlineUserIds" : [ "005D0000001QXi1IAG" ],
  "ChannelName" : "/u/notifications/ExampleUserChannel"
}
```

Using a POST request to `services/data/v29.0/subjects/StreamingChannel/0M6D00000000g7KXA/push` with a request JSON body of:

```
{
  "pushEvents": [
    {
      "payload": "hello world!",
      "userIds": [ "005xx000001Svq3AAC", "005xx000001Svq4AAC" ]
    },
    {
      "payload": "broadcast to everybody (empty user list)!",
      "userIds": []
    }
  ]
}
```

the JSON response data looks something like:

```
[
  {
    "fanoutCount" : 1,
    "userOnlineStatus" : {
      "005xx000001Svq3AAC" : true,
      "005xx000001Svq4AAC" : false,
    }
  },
  {
    "fanoutCount" : -1,
    "userOnlineStatus" : {
    }
  }
]
```

```
}  
]
```

## CHAPTER 6 Monitoring Event Usage

### In this chapter ...

- [Monitor PushTopic Event Usage in the UI](#)
- [Monitor PushTopic and Generic Event Usage with the REST API](#)

Obtain basic daily event usage for PushTopic events through the UI, or full usage information for PushTopic and generic events through the API.

### SEE ALSO:

[PushTopic Streaming Allocations](#)

## Monitor PushTopic Event Usage in the UI

When using API version 36.0 and earlier, you can monitor Streaming API daily event usage for PushTopic events on the Company Information page in Setup.

- From Setup, enter *Company Information* in the Quick Find box, then select **Company Information**. PushTopic event usage is displayed with the label Streaming API Events, Last 24 Hours.

When you refresh the Company Information page, the Streaming API Events value can fluctuate slightly. You can ignore these small fluctuations; your allocations are being assessed accurately.

 **Note:** For API version 37.0 and later, usage information is available only through the API, not in the UI.

## Monitor PushTopic and Generic Event Usage with the REST API

Use the REST API `limits` resource to obtain usage information for Streaming API (API version 36.0 and earlier) and Durable Streaming API (API version 37.0 and later).

The usage information that the `limits` resource returns includes:

Limit Label	Description	API Version
DailyDurableGenericStreamingApiEvents	Generic events notifications delivered in the past 24 hours to all CometD clients for Durable Streaming	37.0 and later
DailyDurableStreamingApiEvents	PushTopic event notifications delivered in the past 24 hours to all CometD clients for Durable Streaming	37.0 and later
DurableStreamingApiConcurrentClients	Concurrent CometD clients (subscribers) across all channels and for all event types for Durable Streaming	37.0 and later
DailyGenericStreamingApiEvents	Generic events notifications delivered in the past 24 hours to all CometD clients	36.0 and earlier
DailyStreamingApiEvents	PushTopic event notifications delivered in the past 24 hours to all CometD clients	36.0 and earlier
StreamingApiConcurrentClients	Concurrent CometD clients (subscribers) across all channels and for all event types	36.0 and earlier

### REST API Endpoint

```
/vXX.X/limits/
```

For more information, see [Limits](#) and [List Organization Limits](#) in the [REST API Developer Guide](#).