CHAPTER 1

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 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.

Consider the following applications for Streaming API.

**Applications That Poll Frequently**

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

**General Notification**

Use Streaming API for applications that require general notification of data changes in an org to 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.
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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect</td>
<td>The client connects to the server.</td>
</tr>
<tr>
<td>disconnect</td>
<td>The client disconnects from the server.</td>
</tr>
<tr>
<td>handshake</td>
<td>The client performs a handshake with the server and establishes a long polling connection.</td>
</tr>
<tr>
<td>subscribe</td>
<td>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 handshake method before you can subscribe to a channel.</td>
</tr>
<tr>
<td>unsubscribe</td>
<td>The client unsubscribes from a channel.</td>
</tr>
</tbody>
</table>

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. Change Data Capture is available as a Developer Preview.

### 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`.

---

3
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:** 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.

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 standard-volume events for 24 hours and high-volume events for 72 hours. With API version 37.0 and later, you can retrieve events that are within the retention window through durable streaming.

**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 are not guaranteed to be contiguous for consecutive events. For example, the event following the event with ID 999 can have an ID of 1,025. 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.

This JSON message shows the `replayId` field in the event object for a generic event.

```json
{
    "clientId":"alps4wpe52qytvcvbsko09tacp",
    "data":{
        "event":{
            "createdDate":"2016-03-29T19:05:28.334Z",
            "replayId":55
        },
        "payload":"This is a message."
    },
    "channel":":/u/TestStreaming"
}
```

This JSON message shows the `replayId` field in the event object for a PushTopic event.

```json
{
    "clientId":"2t80j2hcog29sdh9ihjd9643a",
    "data":{
        "event":{
```
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.

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": {
      "CreatedDate": "2017-04-09T18:31:40.517Z",
      "CreatedById": "005D0000001cSZs",
      "Printer_Model__c": "XZO-5",
      "Serial_Number__c": "12345",
      "Ink_Percentage__c": 0.2
    },
    "event": {
      "replayId": 2
    }
  },
  "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": [
          
        ],
        "changeType": "CREATE",
        "changeOrigin": "com.salesforce.core",
        "transactionKey": "001b7375-0086-250e-e6ca-b99bc3a8b69f",
        "sequenceNumber": 1,
        "isTransactionEnd": true,
        "commitTimestamp": 1501010206653,
        "commitNumber": 92847272780,
      }
    },
  },
  "channel": "/message/Low_Ink__e"
}
```
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 24-hour retention period are discarded.

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

<table>
<thead>
<tr>
<th>Replay Option</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replay ID</td>
<td>Subscriber receives all stored events after the event specified by its replayId value and new events.</td>
<td>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.</td>
</tr>
<tr>
<td>-1</td>
<td>Subscriber receives new events that are broadcast after the client subscribes.</td>
<td>We recommend that clients subscribe with the -1 option to receive new event messages. If clients need to get earlier event messages, they can use any other replay option.</td>
</tr>
<tr>
<td>-2</td>
<td>Subscriber receives all events, including past events that are within the retention window and new events.</td>
<td>Catch up on missed events and retrieve all stored events, for example, after a connection failure. Use</td>
</tr>
</tbody>
</table>
UsageDescriptionReplay Option

<table>
<thead>
<tr>
<th>Replay Option</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>this option sparingly. Subscribing with the -2 option when a large number of event messages are stored can slow performance.</td>
</tr>
</tbody>
</table>

To replay events, use the Streaming API endpoint.

https://Salesforce_Instance/cometd/45.0/

Note: 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.

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.

```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**

**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
Java Samples
For a Java client sample that uses the CometD extension, see the Example: Subscribe to and Replay Events Using a Java Client

SEE ALSO:
Bayeux Protocol, CometD, and Long Polling
Clients and Timeouts

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.

<table>
<thead>
<tr>
<th>Feature</th>
<th>PushTopic Event</th>
<th>Change Data Capture Event</th>
<th>Platform Event</th>
<th>Generic Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define a custom schema as strongly typed fields</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
<td>☒</td>
</tr>
<tr>
<td>Include user-defined payloads</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Publish custom events via one or more APIs</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Publish events via Apex</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Publish declaratively using Process Builder and Flow Builder</td>
<td>N/A</td>
<td>N/A</td>
<td>✓</td>
<td>☒</td>
</tr>
<tr>
<td>Publish to specific users</td>
<td>N/A</td>
<td>N/A</td>
<td>☒</td>
<td>✓</td>
</tr>
<tr>
<td>Subscribe via CometD using JavaScript, Java, and other languages</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Subscribe via Apex triggers</td>
<td>☒</td>
<td>☒</td>
<td>✓</td>
<td>☒</td>
</tr>
<tr>
<td>Filter subscriptions</td>
<td>✓</td>
<td>☒</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Receive auto-published event notifications for Salesforce record changes</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Choose the fields to include in event notifications for Salesforce record changes</td>
<td>✓</td>
<td>☒</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Receive a versioned event schema</td>
<td>☒</td>
<td>✓</td>
<td>✓</td>
<td>☒</td>
</tr>
<tr>
<td>Get field-level security</td>
<td>✓</td>
<td>✓</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Get record-sharing support</td>
<td>✓</td>
<td>☒</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Encrypt field data with Shield Platform Encryption</td>
<td>☒</td>
<td>✓</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td>Replay retained event notifications</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Event retention period</td>
<td>1 day</td>
<td>3 days</td>
<td>1 day or 3 days</td>
<td>1 day</td>
</tr>
</tbody>
</table>
High-volume platform events are retained for up to 3 days.

SEE ALSO:

- Platform Events Developer Guide
- Change Data Capture Developer Guide
CHAPTER 2  Code Examples

In this chapter ...

• Example: Subscribe to and Replay Events Using a Java Client
• Example: Subscribe to and Replay Events Using an Aura 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

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.

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
- Generic events

**Important:** EMP Connector is a free, open-source, community-supported tool. Salesforce provides this tool as an example of how to subscribe to events using CometD. To contribute to the EMP Connector project with your own enhancements, submit pull requests to the repository at [https://github.com/forcedotcom/EMP-Connector](https://github.com/forcedotcom/EMP-Connector).

EMP Connector is based on Java 8 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 InvoiceStatement object from the user interface.

Step 2: Create a PushTopic
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.

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

- Java Development Kit 8 (see Java Downloads)
To run the tool from the command line: Apache Maven (see [https://maven.apache.org/index.html](https://maven.apache.org/index.html))

Access to a Developer Edition org

If you are not already a member of the Lightning Platform developer community, go to [developer.salesforce.com/signup](http://developer.salesforce.com/signup) and follow the instructions for signing up for a Developer Edition organization. Even if you already have Enterprise Edition, Unlimited Edition, or Performance Edition, use Developer Edition for developing, staging, and testing your solutions against sample data to protect your organization’s live data. This is especially true for applications that insert, update, or delete data (as opposed to simply reading data).

- The API Enabled permission must be enabled for your Developer Edition org. This permission is enabled by default, but might have been changed by an administrator.
- The Streaming API permission must be enabled.

*Note:* To verify that the API Enabled and Streaming API permissions are enabled, from Setup, enter User Interface in the Quick Find box, then select User Interface.

- 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 InvoiceStatement object from 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.

<table>
<thead>
<tr>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
</tr>
<tr>
<td>Closed</td>
</tr>
<tr>
<td>Negotiating</td>
</tr>
<tr>
<td>Pending</td>
</tr>
</tbody>
</table>

   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 InvoiceStatement object should now have two custom fields.

**Step 2: Create a PushTopic**

Create a PushTopic in the Developer Console. 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**.

```apex
PushTopic pushTopic = new PushTopic();
pushTopic.Name = 'InvoiceStatementUpdates';
pushTopic.Query = 'SELECT Id, Name, Status__c, Description__c FROM Invoice_Statement__c';
pushTopic.ApiVersion = 45.0;
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 (see Java Downloads)
- Eclipse IDE for Java Developers (get a recent version from http://www.eclipse.org/downloads/eclipse-packages/)

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 production instance. To pass in a custom login URL, such as for sandbox or My Domain, use DevLoginExample.java instead. 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.
     ```bash
     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.

   **Note:** 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 LoginExample 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. This command uses the LoginExample class by default, which logs in to your production instance.

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

To pass in a custom login URL, such as for sandbox or My Domain, use the DevLoginExample class. For example, for sandbox, specify `https://test.salesforce.com` for `<login_URL>`.

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

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.

```java
/*
 * Copyright (c) 2016, salesforce.com, inc.
 * All rights reserved.
 * Licensed under the BSD 3-Clause license.
 * For full license text, see LICENSE.TXT file in the repo root or
 * https://opensource.org/licenses/BSD-3-Clause
 */
package com.salesforce.emp.connector.example;

import static com.salesforce.emp.connector.LoginHelper.login;
import java.net.URL;
import java.util.Map;
import java.util.concurrent.TimeUnit;
import java.util.function.Consumer;
import com.salesforce.emp.connector.BayeuxParameters;
import com.salesforce.emp.connector.EmpConnector;
import com.salesforce.emp.connector.LoginHelper;
import com.salesforce.emp.connector.TopicSubscription;

/**
 * An example of using the EMP connector using login credentials
 */
public class LoginExample {
    public static void main(String[] argv) throws Exception {
        if (argv.length < 3 || argv.length > 4) {
            System.err.println("Usage: LoginExample username password channel [replayFrom]");
            System.exit(1);
        }
        long replayFrom = EmpConnector.REPLAY_FROM_EARLIEST;
        if (argv.length == 4) {
            replayFrom = Long.parseLong(argv[3]);
        }

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

EmpConnector connector = new EmpConnector(params);

connector.setBearerTokenProvider(tokenProvider);

connector.start().get(5, TimeUnit.SECONDS);

TopicSubscription subscription = connector.subscribe(
    argv[2], replayFrom, consumer).get(5, TimeUnit.SECONDS);

System.out.println(String.format("Subscribed: %s", subscription));
}

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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>Username of the logged-in user</td>
</tr>
<tr>
<td>password</td>
<td>Password for the username (or logged-in user)</td>
</tr>
<tr>
<td>channel</td>
<td>The channel name for the event: /event/Low_Ink__e</td>
</tr>
</tbody>
</table>

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
   /event/EventName__e

Change Data Capture event
   For all change events—/data/ChangeEvents
   For a specific standard object—/data/ObjectNameChangeEvent
   For a specific custom object—/data/CustomObjectName__ChangeEvent

PushTopic event
   /topic/PushTopicName

Generic event
   /u/notifications/GenericStreamingChannel

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.
The sample fetches the earliest saved events within the past 24 hours. 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.
- 2—Get all new events sent after subscription and all past events within the past 24 hours.
- 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:-2]
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, do not 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.

**Prerequisites**

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

See Set Up Authentication with OAuth 2.0. Also see Authenticate Apps with OAuth in Salesforce Help and Understanding Authentication in the REST API Developer Guide.

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.

```java
/*
 * Copyright (c) 2016, salesforce.com, inc. All rights reserved. Licensed under the BSD 3-Clause license. For full
 * license text, see LICENSE.TXT file in the repo root or
 * https://opensource.org/licenses/BSD-3-Clause
 */
package com.salesforce.emp.connector.example;

import java.net.MalformedURLException;
import java.net.URL;
```
import java.util.Map;
import java.util.concurrent.TimeUnit;
import java.util.function.Consumer;

import com.salesforce.emp.connector.BayeuxParameters;
import com.salesforce.emp.connector.EmpConnector;
import com.salesforce.emp.connector.TopicSubscription;
import org.cometd.bayeux.Channel;

/**
 * An example of using the EMP connector using bearer tokens
 */
public class BearerTokenExample {
    public static void main(String[] argv) throws Exception {
        if (argv.length < 2 || argv.length > 4) {
            System.err.println("Usage: BearerTokenExample url token channel
[replayFrom]"");
            System.exit(1);
        }
        long replayFrom = EmpConnector.REPLAY_FROM_EARLIEST;
        if (argv.length == 4) {
            replayFrom = Long.parseLong(argv[3]);
        }

        BayeuxParameters params = new BayeuxParameters() {
            @Override
            public String bearerToken() {
                return argv[1];
            }

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

        Consumer<Map<String, Object>> consumer = event -> System.out.println(
            String.format("Received:
%s", event));
        EmpConnector connector = new EmpConnector(params);

        connector.addListener(Channel.META_CONNECT, new LoggingListener(true, true))
            .addListener(Channel.META_DISCONNECT, new LoggingListener(true, true))
            .addListener(Channel.META_HANDSHAKE, new LoggingListener(true, true));

        connector.start().get(5, TimeUnit.SECONDS);

        TopicSubscription subscription = connector.subscribe(
            argv[2], replayFrom, consumer).get(5, TimeUnit.SECONDS);
System.out.println(String.format("Subscribed: %s", subscription));
}
}

2. Run the BearerTokenExample class, and provide the following argument values.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>URL of the Salesforce instance of the logged-in user</td>
</tr>
<tr>
<td>token</td>
<td>The access token returned by the OAuth authentication flow</td>
</tr>
<tr>
<td>channel</td>
<td>The channel name for the event: /event/Low_Ink__e</td>
</tr>
</tbody>
</table>

**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**
/event/EventName__e

**Change Data Capture event**
- For all change events—/data/ChangeEvents
- For a specific standard object—/data/ObjectNameChangeEvent
- For a specific custom object—/data/CustomObjectName__ChangeEvent

**PushTopic event**
/topic/PushTopicName

**Generic event**
/u/notifications/GenericStreamingChannel

The sample fetches the earliest saved events within the past 24 hours. 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.
- `-2`—Get all new events sent after subscription and all past events within the past 24 hours.
- 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.

```plaintext
Subscribed: Subscription [/topic/InvoiceStatementUpdates:-2]
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},
```
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.

```java
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.

```java
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.

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

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

// 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.

```java
// 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 an Aura Component

Add the lightning:empApi component to your custom Aura component to receive event messages. The lightning: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
• Generic events

The `lightning:empApi` component uses a shared CometD-based Streaming API connection, enabling you to run multiple streaming apps in the browser.

To call the component’s methods, add the `lightning:empApi` component inside your custom component and assign an `aura:id` attribute to it.

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

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

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

**Note:** As of API version 45.0, you can build Lightning components using two programming models: Lightning Web Components, and the original model, Aura Components. 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. This developer guide covers Aura components only.

---

**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
- Deploy a Sample Project to Your Org
- Durable PushTopic Streaming Sample
- Durable Generic Streaming Sample
- Replay Events Sample: Code Walkthrough

**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. The “API Enabled” permission is enabled by default, but an administrator might have changed it.
If you are not already a member of the Lightning Platform developer community, go to developer.salesforce.com/signup and follow the instructions for signing up for a Developer Edition organization. Even if you already have Enterprise Edition, Unlimited Edition, or Performance Edition, use Developer Edition for developing, staging, and testing your solutions against sample data to protect your organization’s live data. This is especially true for applications that insert, update, or delete data (as opposed to simply reading data).

Note: To verify that the “API Enabled” and “Streaming API” permissions are enabled in your organization, from Setup, enter User Interface in the Quick Find box, then select User Interface.

- 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 Workbench to copy all project components to your org.

1. Download the Salesforce Durable Streaming Demo .zip file from the developerforce github repository.

   If you want, 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. The following common components are installed in your org when you deploy the .zip file.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cometdReplayExtension</td>
<td>Static resource representing a CometD extension in JavaScript. This extension implements the replay mechanism for Streaming API.</td>
</tr>
<tr>
<td>cometd, jquery, jquery_cometd, json2</td>
<td>Static resources for CometD 3.1.0, jquery, and JSON.</td>
</tr>
</tbody>
</table>

The following app components are for the Durable PushTopic Streaming page.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DurablePushTopicEventDisplay</td>
<td>A Visualforce component that uses the CometD extension cometdReplayExtension 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.</td>
</tr>
<tr>
<td>DurablePushTopicStreamingController</td>
<td>Apex controller that holds the logic behind the Visualforce page.</td>
</tr>
<tr>
<td>DurablePushTopicStreamingDemo Visualforce Page</td>
<td>Visualforce page. This page is the main page you use to generate, view, and replay durable PushTopic events.</td>
</tr>
</tbody>
</table>

The following app components are for the Durable Generic Streaming page.
A Visualforce component that uses the CometD extension `cometdReplayExtension` 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.

**Component** | **Description**
--- | ---
**DurableGenericEventDisplay** | A Visualforce component that uses the CometD extension `cometdReplayExtension` 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.

**DurableGenericStreamingController** | Apex controller that holds the logic behind the Visualforce page.

**StreamingChannel** | Custom object used for creating streaming channels.

**DurableGenericStreamingDemo Visualforce Page** | Visualforce page. This page is the main page you use to generate, view, and replay durable generic events.

**DurableStreamingDemo Permission Set** | Permission set used to grant read and create access to the StreamingChannel sObject.

You use Workbench to migrate the zip file to your org.


⚠️ **Important:** Workbench is a free, open source, community-supported tool (see the Help page in Workbench). Salesforce provides a hosted instance of Workbench for demonstration purposes only—Salesforce recommends that you do not use this hosted instance of Workbench to access data in a production database. If you want to use Workbench for your production database, you can download, host, and configure it using your own resources. You can download Workbench from [https://github.com/forceworkbench/forceworkbench](https://github.com/forceworkbench/forceworkbench).

3. For **Environment**, keep the production default value.

4. Ensure that the value for **API Version** is at least 37.0.

5. Accept the service terms, and then click **Login with Salesforce**.

6. Enter your Developer Edition org username and password, and then click **Log In**.

7. Select **migration > Deploy**.

8. Click **Choose File** and locate the .zip file that you downloaded.

9. Click **Next**, and then click **Deploy**.

10. Wait until the deployment finishes and the status of the deployment changes to **Succeeded**.

11. Log in to your org in another browser tab.

**SEE ALSO:**

[GitHub: Streaming Replay Client Extensions](https://github.com/forceworkbench/forceworkbench)

### 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. 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.
3. On the Visualforce page, generate some events for an account. For example, Test account.
4. 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.

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 2, enter 2. Then click Update Subscription.
   The Notifications section is updated and shows only the last event with replay ID 3.
6. 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.

7. 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.
8. Switch the replay option back to -2.
The page displays all events, including events that were sent earlier.
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 programatically using Apex or a data API like the SOAP API or REST API, or using a tool such as Workbench. 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. 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.
6. 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.

7. 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.
8. Switch the replay option back to -2.
   The page displays all events, including events that were sent earlier.
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.

```javascript
// 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.

```javascript
// 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.

```javascript
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".

```javascript
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 comet 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.

```javascript
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.

```javascript
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.
Prerequisites

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

- Access to a Developer Edition organization.
  
  If you are not already a member of the Lightning Platform developer community, go to developer.salesforce.com/signup and follow the instructions for signing up for a Developer Edition organization. Even if you already have Enterprise Edition, Unlimited Edition, or Performance Edition, use Developer Edition for developing, staging, and testing your solutions against sample data to protect your organization's live data. This is especially true for applications that insert, update, or delete data (as opposed to simply reading data).

- The “API Enabled” permission must be enabled for your Developer Edition organization. This permission is enabled by default, but may have been changed by an administrator.

- The “Streaming API” permission must be enabled.

  Note: To verify that the “API Enabled” and “Streaming API” permissions are enabled in your organization, from Setup, enter User Interface in the Quick Find box, then select User Interface.

- 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 InvoiceStatement object from the user interface.

After you create a PushTopic and subscribe to it, you’ll get notifications when an InvoiceStatement 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.

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
</tr>
<tr>
<td>Closed</td>
</tr>
<tr>
<td>Negotiating</td>
</tr>
<tr>
<td>Pending</td>
</tr>
</tbody>
</table>

e. Select the checkbox for **Use first value as default value**.

f. 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 InvoiceStatement 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**.

```apex
PushTopic pushTopic = new PushTopic();
pushTopic.Name = 'InvoiceStatementUpdates';
pushTopic.Query = 'SELECT Id, Name, Status__c, Description__c FROM Invoice_Statement__c';
pushTopic.ApiVersion = 45.0;
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:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cometd.zip</td>
<td>Files for CometD version 3.1.0 and the Dojo toolkit used by demo.js. 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.</td>
</tr>
<tr>
<td>demo.css</td>
<td>The CSS code that formats the Visualforce page.</td>
</tr>
<tr>
<td>demo.js</td>
<td>The code used by the page to subscribe to the channel, receive and display the notifications, and unsubscribe from the channel.</td>
</tr>
<tr>
<td>json2.js</td>
<td>The JavaScript library that contains the stringify and parse methods.</td>
</tr>
<tr>
<td>StreamingApiDemo</td>
<td>The Visualforce page that displays the Streaming API notifications.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>File Name</th>
<th>Static Resource Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>cometd.zip</td>
<td>cometd_zip</td>
</tr>
<tr>
<td>demo.css</td>
<td>demo_css</td>
</tr>
<tr>
<td>demo.js</td>
<td>demo_js</td>
</tr>
</tbody>
</table>
For more information about static resources, see *Deliver Static Resources with Visualforce.*

**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.

```html
<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="unsubscribeButton" class="button">Unsubscribe</button>
          </td>
        </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
</script>
```
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 InvoiceStatement in a different browser tab. The page displays some debug messages and event notifications. The output resembles the following:

   ```json
   DEBUG: Handshake Successful: {
   "ext": {"replay":true,"payload.format":true},
   "minimumVersion": "1.0",
   "clientId": "41kdiuvig2txdh9weakuiwyh",
   "supportedConnectionTypes": ["long-polling"],
   "channel": "/meta/handshake",
   "id": "1",
   "version": "1.0",
   "successful": true,
   "reestablish": false}
   ```

   ```json
   DEBUG: Connection Successful : {
   "clientId": "41kdiuvig2txdh9weakuiwyh",
   "advice": {
   "interval": 2000,
   "multiple-clients": true,
   "reconnect": "retry",
   "timeout": 110000,
   "channel": "/meta/connect",
   "id": "2",
   "successful": true}
   ```

   ```json
   DEBUG: Subscribe Successful /topic/InvoiceStatementUpdates: {
   "clientId": "41kdiuvig2txdh9weakuiwyh",
   "channel": "/meta/subscribe",
   "id": "15",
   "subscription": "/topic/InvoiceStatementUpdates",
   "successful": true}
   ```

   ```json
   {
   "event": {
   "createdDate": "2017-05-16T23:05:50.173Z",
   "replayId": 1,
   "type": "created"
   },
   "object": {
   "Description__c": "New invoice."
   }
   }
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 Authentication 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 Authentication 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 may 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. Usually it’s a servlet if you work with Java. 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 click Save, the Consumer Key is created and displayed, and a Consumer Secret is created (click the link to reveal it).

   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 Digging Deeper into OAuth 2.0 in Salesforce.

   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:

   ```java
   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: This method of authentication should only be used in development environments and not for production code.

   Example: This example gets the session ID (authenticates), and then follows a resource, https://instance.salesforce.com/id/00Dxxxxxxxxxxxx/005xxxxxxxxxxxx contained in the first response to get more information about the user.

   ```java
   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 + "...
");
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("\n");

// 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("\n");

// 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}", "45.0"));
}

The output from this code resembles the following:

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

OAuth login response
  id = https://login.salesforce.com/id/00D30000000ehjIEAQ/00530000003THy8AAG
  issued_at = 1334961666037
  instance_url = https://instance.salesforce.com
  access_token = 00D30000000ehjIARYAQHc.0Mlmz.DCg3HRNF.SmsSn5njPkry2SM6p6rjCQqfAO2v5CGksRSPRb.xb
  signature = 8M9VWBoaEk+Bs/yD+BfrUR/+5tkNLgXAIwallPMwsY=

User info response
  user_type = STANDARD
  status = {created_date=2012-04-08T16:44:58.000+0000, body=Hello}
  urls = {sobjects=https://instance.salesforce.com/services/data/v{version}/sobjects/,
         feeds=https://instance.salesforce.com/services/data/v{version}/chatter.feeds,
         users=https://instance.salesforce.com/services/data/v{version}/chatter/users,
         query=https://instance.salesforce.com/services/data/v{version}/query/,
         enterprise=https://instance.salesforce.com/services/Soap/c/{version}/00D30000000ehjI,
         recent=https://instance.salesforce.com/services/data/v{version}/recent/,
         feed_items=https://instance.salesforce.com/services/data/v{version}/chatter/feed-items,
         search=https://instance.salesforce.com/services/data/v{version}/search/,
         partner=https://instance.salesforce.com/services/Soap/u/{version}/00D30000000ehjI,
         rest=https://instance.salesforce.com/services/data/v{version}/,
         groups=https://instance.salesforce.com/services/data/v{version}/chatter/groups,
         metadata=https://instance.salesforce.com/services/Soap/m/{version}/00D30000000ehjI,
         profile=https://instance.salesforce.com/00530000003THy8AAG
  locale = en_US
  asserted_user = true
  id = https://login.salesforce.com/id/00D30000000ehjIEAQ/00530000003THy8AAG
  nick_name = SampleNickname
  photos = {picture=https://instance.content.force.com/profilephoto/005/F,
            thumbnail=https://instance.content.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://instance.salesforce.com/services/data/v45.0/
CHAPTER 3  
Client Connection Considerations

In this chapter ...

- Clients and Timeouts
- Clients and Cookies for Streaming API
- Supported CometD Versions
- HTTPS Recommended
- Debugging Streaming API Applications

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 server closes the connection. Clients should reconnect before two minutes to avoid the connection 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://instance_name.salesforce.com/cometd.

Streaming API requirements on clients:

- The "Content-Type: application/json" header is required on all calls to the cometd servlet if the content of the post is JSON.
- 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 might 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/.
HTTPS Recommended

Streaming API follows the preference set by your administrator for your organization. By default this is HTTPS. To protect the security of your data, we recommend you use HTTPS.

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 is not 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 is not 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.

```json
{
    "clientId": "1qlib66fvm7kli1gfoauu95i78g",
    "advice": {
        "reconnect": "none",
        "interval": 0
    },
    "channel": "/meta/connect",
    "id": 7,
    "error": "401::Authentication invalid"
}
```
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 EMPConnector 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.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Message</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>API version in the URI is mandatory. URI format: '/cometd/45.0'</td>
<td>The API version information was not in the URL. Include the API version at the end of the URI. For example, /cometd/45.0.</td>
</tr>
<tr>
<td>400</td>
<td>Unsupported API version. Only API versions '23.0' and above are supported. URI format: '/cometd/45.0'</td>
<td>The supplied API version in the URI is not supported. Only API version 23.0 and later is supported. The URI format is /cometd/xx.x.</td>
</tr>
<tr>
<td>400</td>
<td>Invalid connection type {connection_type}</td>
<td>An invalid transport type was used. Only long-polling is supported, but another connection type was requested, such as WebSocket or callback long-polling.</td>
</tr>
<tr>
<td>400</td>
<td>The channel you requested to subscribe to does not exist {channel_name}</td>
<td>The streaming channel requested to subscribe to doesn’t exist. Ensure that the channel is created before subscribing.</td>
</tr>
<tr>
<td>400</td>
<td>Channel name not specified</td>
<td>The channel name wasn’t specified. Specify a valid channel name to subscribe to.</td>
</tr>
<tr>
<td>400</td>
<td>Channel subscriptions must start with a leading '/'</td>
<td>The channel name format is invalid. Channel names must start with a leading slash (/).</td>
</tr>
<tr>
<td>400</td>
<td>Query fields {query_fields} do not exist on the topic entity</td>
<td>The supplied query fields don’t exist on the Salesforce object specified in the PushTopic.</td>
</tr>
<tr>
<td>400</td>
<td>Client client_name has established a session, but no cookie_name cookie present</td>
<td>No cookie was found after the client established a session. Ensure that the streaming client accepts cookies.</td>
</tr>
<tr>
<td>400</td>
<td>The replayId {replay_id} you provided was invalid. Please provide a valid ID, -2 to replay all events, or -1 to replay only new events.</td>
<td>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.</td>
</tr>
<tr>
<td>401</td>
<td>Authentication invalid.</td>
<td>The supplied authentication token or session ID is not valid. This error is returned on the /meta/handshake or the /meta/connect channel. On the /meta/handshake channel, the error is in the failureReason response field, which is nested under the ext/sfdc field. On the /meta/connect channel, the error is in the error field.</td>
</tr>
<tr>
<td>401</td>
<td>Request requires authentication.</td>
<td>No authentication token or session ID was supplied in the request header. The client must send authentication</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Message</td>
<td>Error Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>403</td>
<td>Cannot create channel <code>{channel_name}</code></td>
<td>The subscription channel can't be created, which can be due to insufficient permissions.</td>
</tr>
<tr>
<td>403</td>
<td>Subscriber does not have access to the entity in this topic</td>
<td>The subscriber doesn't have access to the Salesforce object in the PushTopic.</td>
</tr>
<tr>
<td>403</td>
<td>Subscriber does not have access to all fields referenced in the where clause of the PushTopic</td>
<td>The subscriber doesn't have access to all fields referenced in the WHERE clause of the PushTopic.</td>
</tr>
<tr>
<td>403</td>
<td>Handshake denied</td>
<td>The handshake request was denied. The cause of this error is provided in the failureReason field in the response, which is nested under the ext/sfdc field.</td>
</tr>
<tr>
<td>403</td>
<td>Client has not completed handshake</td>
<td>The client has not completed a handshake. The client must perform a handshake before subscribing.</td>
</tr>
<tr>
<td>403</td>
<td>Organization concurrent user limit exceeded</td>
<td>The maximum number of concurrent clients across all channels has been exceeded. This error applies to any type of event, including PushTopic, generic, and platform events.</td>
</tr>
<tr>
<td>403</td>
<td>Organization total events daily limit exceeded</td>
<td>The maximum number of daily events has been exceeded. This error applies to any type of event, including PushTopic, generic, and platform events.</td>
</tr>
<tr>
<td>403</td>
<td>Restricted channel</td>
<td>The user doesn't have the required permissions to subscribe to the streaming channel.</td>
</tr>
<tr>
<td>403</td>
<td>User not enabled for streaming</td>
<td>The user doesn't have read permission on the PushTopic.</td>
</tr>
<tr>
<td>403</td>
<td>User not allowed to subscribe CDC without View All Data permissions</td>
<td>The user must have the View All Data permission to subscribe to Change Data Capture.</td>
</tr>
<tr>
<td>403</td>
<td>Subscription limit exceeded for this topic</td>
<td>The maximum number of concurrent clients per topic for PushTopic and generic events has been exceeded. This error doesn't apply to platform events.</td>
</tr>
<tr>
<td>403</td>
<td>Unknown client</td>
<td>The server deleted the client CometD session due to a timeout, which can be caused by a network failure. The client must perform a new handshake and reconnect.</td>
</tr>
<tr>
<td>403</td>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Error Description

- *(on the /meta/handshake channel)* in the `failureReason` response field, which is nested under the `ext/sfdc` field. The response also contains an error in the `error` field: 403::Handshake denied.

- Maximum Request Size Exceeded
  - The maximum request size of 32,768 bytes has been exceeded.

- 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 `failureReason` response field, which is nested in the `ext/sfdc` field on the /meta/handshake or /meta/connect channels. The `error` field in the response contains another error alongside this error.

### Generic Streaming-only Errors

The following errors are returned for generic streaming events only.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Message</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>403</td>
<td>Unable to create channel dynamically, maximum channel limit has been exceeded</td>
<td>The maximum number of generic streaming channels has been exceeded.</td>
</tr>
<tr>
<td>403</td>
<td>No access on channel</td>
<td>The generic streaming channel can’t be accessed because the user doesn’t have permissions on the StreamingChannel object.</td>
</tr>
<tr>
<td>404</td>
<td>channel names may not vary only by case</td>
<td>The generic streaming channel exists with a different case. Generic streaming channel names are case-sensitive.</td>
</tr>
<tr>
<td>404</td>
<td>Unknown channel</td>
<td>The generic streaming channel isn’t found or can’t be created dynamically.</td>
</tr>
</tbody>
</table>
CHAPTER 4  PushTopic Events

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

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.

Note: Updates performed by the Bulk API won’t generate notifications, since such updates could flood a channel.

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 time may fluctuate depending on the overall server load.

The PushTopic defines when notifications are generated in the channel. This is specified by configuring the following PushTopic fields:

• PushTopic Queries
• Events
• Notifications

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

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 InvoiceStatement__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:

```json
{
    "channel": "/topic/InvoiceStatementUpdates",
    "data": {
        "event": {
            "type": "updated",
            "createdDate": "2011-11-03T15:59:06.000+0000"
        },
        "sobject": {
            "Name": "INV-0001",
            "Id": "a00D0000008y8IAA",
            "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.

**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 specific 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 will not receive the notification.

For example, assume a user tries to subscribe to a PushTopic with the following *Query* value:

```sql
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 won’t be returned in the notification.

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

**Supported PushTopic Queries**

All custom objects are supported in PushTopic queries. The following subset of standard objects are supported in PushTopic queries.

- Account
- Campaign
- Case
- Contact
- ContractLineItem
- Entitlement
- Lead
- LiveChatTranscript
- Opportunity
- Quote
- QuoteLineItem
- ServiceAppointment
- ServiceContract
- Task
- WorkOrder
- WorkOrderLineItem
**Important:** Tasks that are created or updated using the following 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 as well as 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**
  
  ```sql
  SELECT Id, MyCustomField__c FROM MyCustomObject__c
  ```

- **Standard objects (may include custom fields)**
  
  - **Account**
    
    ```sql
    SELECT Id, Name FROM Account WHERE NumberOfEmployees > 1000
    ```

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

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

  - **Contact**
    
    ```sql
    SELECT Id, Name, Email FROM Contact;
    ```

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

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

  - **Task**
    
    ```sql
    SELECT Id, Subject, IsClosed, Status FROM Task WHERE isClosed = TRUE
    ```

**Important:**

- 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.
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.

<table>
<thead>
<tr>
<th>Fields</th>
<th>Supported?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT Id, Name</td>
<td>Yes</td>
</tr>
<tr>
<td>SELECT Id, Name, firstName, lastName</td>
<td>Yes</td>
</tr>
<tr>
<td>SELECT Id, firstName, lastName</td>
<td>No</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Fields</th>
<th>Supported?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT Id, MailingAddress</td>
<td>No</td>
</tr>
<tr>
<td>SELECT Id, MailingAddress, MailingCity, MailingStreet</td>
<td>Yes</td>
</tr>
<tr>
<td>SELECT Id, MailingCity, MailingStreet</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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.
Unsupported PushTopic Queries

The following SOQL statements are not 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 (formula fields are supported in SELECT clauses though.)

- **NOT**
PushTopic Events

- 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

**Note:** TYPEOF is currently available as a Developer Preview as part of the SOQL Polymorphism feature. For more information on enabling TYPEOF for your organization, contact Salesforce.

---

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:

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

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:
### NotifyForOperations Value

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All (default)</td>
<td>Evaluate a record to possibly generate a notification whether the record has been created or updated.</td>
</tr>
<tr>
<td>Create</td>
<td>Evaluate a record to possibly generate a notification only if the record has been created.</td>
</tr>
<tr>
<td>Update</td>
<td>Evaluate a record to possibly generate a notification only if the record has been updated.</td>
</tr>
<tr>
<td>Extended</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>NotifyForFields Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Notifications are generated for all record field changes, provided the evaluated records match the criteria specified in the WHERE clause.</td>
</tr>
<tr>
<td>Referenced (default)</td>
<td>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.</td>
</tr>
<tr>
<td>Select</td>
<td>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.</td>
</tr>
<tr>
<td>Where</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Event</th>
<th>A notification is generated when</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record is created</td>
<td>The record field values match the values specified in the WHERE clause.</td>
</tr>
<tr>
<td>Record is updated</td>
<td>The record field values match the values specified in the WHERE clause.</td>
</tr>
</tbody>
</table>
### Examples

<table>
<thead>
<tr>
<th>PushTopic Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT Id, f1, f2, f3 FROM InvoiceStatement</code></td>
<td>Generates a notification if any field values in the record have changed.</td>
</tr>
<tr>
<td><code>SELECT Id, f1, f2 FROM InvoiceStatement WHERE f3 = 'abc' AND f4 LIKE 'xyz'</code></td>
<td>Generates a notification if any field values in the record have changed and f3 and f4 match the values in the WHERE clause.</td>
</tr>
<tr>
<td><code>SELECT Id FROM InvoiceStatement WHERE f3 = 'abc' AND f4 LIKE 'xyz'</code></td>
<td>When Id is the only field in the SELECT clause, a notification is generated if any field values have changed.</td>
</tr>
<tr>
<td><code>SELECT Id FROM InvoiceStatement WHERE f3 = 'abc' AND f4 LIKE 'xyz'</code></td>
<td>Generates a notification if any field values in the record have changed and f3 and f4 match the values in the WHERE clause.</td>
</tr>
<tr>
<td><code>SELECT Id FROM InvoiceStatement WHERE Id IN ('a07B0000000KWZ7IAO', 'e10R0000000KEU9IAO', 'v32B0000000KWZ7YEP')</code></td>
<td>Generates a notification if any field values in the record have changed and the record ID is contained in the WHERE clause IN list.</td>
</tr>
<tr>
<td><code>SELECT Id, f1, f2 FROM InvoiceStatement WHERE Id IN ('a07B0000000KWZ7IAO', 'e10R0000000KEU9IAO', 'v32B0000000KWZ7YEP')</code></td>
<td>Generates a notification if any field values in the record have changed and the record ID is contained in the WHERE clause IN list.</td>
</tr>
<tr>
<td><code>SELECT Id, f1, f2 FROM InvoiceStatement WHERE f3 = 'abc' AND f4 LIKE 'xyz' AND Id IN ('a07B0000000KWZ7IAO', 'e10R0000000KEU9IAO', 'v32B0000000KWZ7YEP')</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

⚠️ **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`.

<table>
<thead>
<tr>
<th>Event</th>
<th>A notification is generated when</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record is created</td>
<td>The record field values match the values specified in the <code>WHERE</code> clause.</td>
</tr>
<tr>
<td>Record is updated</td>
<td>The following conditions are met.</td>
</tr>
<tr>
<td></td>
<td>• A change occurs in one or more record fields that are specified in the <code>PushTopic</code> query <code>SELECT</code> or <code>WHERE</code> clause.</td>
</tr>
<tr>
<td></td>
<td>• For fields specified in the <code>WHERE</code> clause, all record field values match the values in the <code>WHERE</code> clause.</td>
</tr>
</tbody>
</table>

**Examples**

<table>
<thead>
<tr>
<th>PushTopic Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT Id, f1, f2, f3 FROM InvoiceStatement__c</code></td>
<td>Generates a notification if `f1, f2, or f3 have changed.</td>
</tr>
<tr>
<td><code>SELECT Id, f1, f2 FROM InvoiceStatement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'</code></td>
<td>Generates a notification if <code>f1, f2, f3, or f4 have changed and </code>f3 and f4 match the values in the <code>WHERE</code> clause.`</td>
</tr>
<tr>
<td><code>SELECT Id FROM InvoiceStatement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'</code></td>
<td>Generates a notification if <code>f3 and f4 have changed and f3 and f4 match the values in the </code>WHERE<code> clause.</code></td>
</tr>
<tr>
<td><code>SELECT Id, f1, f2 FROM InvoiceStatement__c WHERE Id IN ('a07B0000000KZ7IAO', 'e10R0000000KEU9IAO', 'v32B0000000KW7YEP')</code></td>
<td>Generates a notification if <code>f1 or f2 have changed and the record ID is contained in the </code>WHERE<code> clause IN list.</code></td>
</tr>
<tr>
<td><code>SELECT Id, f1, f2 FROM InvoiceStatement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz' AND Id IN ('a07B0000000KZ7IAO', 'e10R0000000KEU9IAO', 'v32B0000000KW7YEP')</code></td>
<td>Generates a notification if <code>f1, f2, f3, or f4 have changed, f3 and f4 match the values in the </code>WHERE<code>clause, and the ID is contained in the</code>WHERE<code> clause IN list.</code></td>
</tr>
</tbody>
</table>
**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`.

<table>
<thead>
<tr>
<th>Event</th>
<th>A notification is generated when</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record is created</td>
<td>The record field values match the values specified in the <code>WHERE</code> clause.</td>
</tr>
<tr>
<td>Record is updated</td>
<td>The following conditions are met.</td>
</tr>
<tr>
<td></td>
<td>• A change occurs in one or more record fields specified in the <code>PushTopic</code> query <code>SELECT</code> clause.</td>
</tr>
<tr>
<td></td>
<td>• The record values of the fields specified in the <code>WHERE</code> clause all match the values in the <code>PushTopic</code> query <code>WHERE</code> clause.</td>
</tr>
</tbody>
</table>

**Examples**

<table>
<thead>
<tr>
<th>PushTopic Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT Id, f1, f2, f3 FROM InvoiceStatement__c</code></td>
<td>Generates a notification if <code>f1, f2, or f3</code> have changed.</td>
</tr>
<tr>
<td><code>SELECT Id, f1, f2 FROM InvoiceStatement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'</code></td>
<td>Generates a notification if <code>f1 or f2</code> have changed and <code>f3</code> and <code>f4</code> match the values in the <code>WHERE</code> clause.</td>
</tr>
<tr>
<td><code>SELECT Id, f1, f2 FROM InvoiceStatement__c WHERE Id IN ('a07B0000000KWZ7IAO', 'e10R0000000KEU9IAO', 'v32B0000000KWZ7YPE')</code></td>
<td>Generates a notification if <code>f1 or f2</code> have changed and <code>Id</code> is contained in the <code>WHERE</code> clause <code>IN</code> list.</td>
</tr>
<tr>
<td><code>SELECT Id, f1, f2 FROM InvoiceStatement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz' AND Id IN ('a07B0000000KWZ7IAO', 'e10R0000000KEU9IAO', 'v32B0000000KWZ7YPE')</code></td>
<td>Generates a notification if <code>f1 or f2</code> have changed, <code>f3</code> and <code>f4</code> match the values in the <code>WHERE</code> clause, and the <code>Id</code> is contained in the <code>WHERE</code> clause <code>IN</code> list.</td>
</tr>
</tbody>
</table>

**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 Notification Rules**

### Event Notification Rules

<table>
<thead>
<tr>
<th>Event</th>
<th>A notification is generated when</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record is created</td>
<td>The record field values match the values specified in the WHERE clause.</td>
</tr>
<tr>
<td>Record is updated</td>
<td>The following conditions are met.</td>
</tr>
<tr>
<td></td>
<td>• A change occurs in one or more record fields specified in the PushTopic query WHERE clause.</td>
</tr>
<tr>
<td></td>
<td>• The record values of the fields specified in the WHERE clause all match the values in the PushTopic query WHERE clause.</td>
</tr>
</tbody>
</table>

### Examples

<table>
<thead>
<tr>
<th>PushTopic Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT Id, f1, f2 FROM InvoiceStatement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'</td>
<td>Generates a notification if f3 or f4 have changed and the values match the values in the WHERE clause.</td>
</tr>
<tr>
<td>SELECT Id FROM InvoiceStatement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz'</td>
<td>Generates a notification if f3 or f4 have changed and the values match the values in the WHERE clause.</td>
</tr>
<tr>
<td>SELECT Id, f1, f2 FROM InvoiceStatement__c WHERE f3 = 'abc' AND f4 LIKE 'xyz' AND Id IN ('a07B0000000KWZ7IAO', 'e10R0000000KEU9IAO', 'v32B0000000KWZ7YEP')</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Notification Scenarios

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

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>You want to receive all notifications of all record updates.</td>
<td>• MyPushTopic.Query = SELECT Id, Name, Description__c FROM InvoiceStatement</td>
</tr>
<tr>
<td></td>
<td>• MyPushTopic.NotifyForFields = All</td>
</tr>
<tr>
<td>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.</td>
<td>• MyPushTopic.Query = SELECT Id, Name, Amount__c FROM InvoiceStatement</td>
</tr>
<tr>
<td></td>
<td>• MyPushTopic.NotifyForFields = Referenced</td>
</tr>
<tr>
<td>You want to receive notification of all record changes made to a specific record.</td>
<td>• MyPushTopic.Query = SELECT Id, Name, Amount__c FROM InvoiceStatement WHERE Id='a07B0000000KWZ7IAO'</td>
</tr>
<tr>
<td></td>
<td>• MyPushTopic.NotifyForFields = All</td>
</tr>
</tbody>
</table>
### Scenario

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.

<table>
<thead>
<tr>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MyPushTopic</strong>.Query = SELECT Id, Name, Amount__c FROM InvoiceStatement WHERE Id='a07B0000000KWZ7IAO'</td>
</tr>
<tr>
<td><strong>MyPushTopic</strong>.NotifyForFields = Referenced</td>
</tr>
</tbody>
</table>

You want to receive notification for all invoice statement record changes for vendors in a particular state.

<table>
<thead>
<tr>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MyPushTopic</strong>.Query = SELECT Id, Name, Amount__c FROM InvoiceStatement WHERE BillingState__c = 'NY'</td>
</tr>
<tr>
<td><strong>MyPushTopic</strong>.NotifyForFields = All</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MyPushTopic</strong>.Query = SELECT Id, Name FROM InvoiceStatement WHERE Amount &gt; 999</td>
</tr>
<tr>
<td><strong>MyPushTopic</strong>.NotifyForFields = Referenced</td>
</tr>
</tbody>
</table>

---

### 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/ChannelName?<expression>
```

*ChannelName* is the channel, 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.

**Note:** If you use an ID in filter criteria, use the 18-character ID format; 15-character IDs aren’t supported.

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

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

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

```json
[
  {
    "channel": "/meta/subscribe",
    "clientId": "Uniq1dentifier",
    "subscription": "/topic/foo"
  },
  {
    "channel": "/meta/subscribe",
    "clientId": "Uniq1dentifier",
    "subscription": "/topic/bar"
  },
  {
    "channel": "/meta/subscribe",
    "clientId": "Uniq1dentifier",
    "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:

  ```apex
  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:

  - Notification Message Order
  - Considerations for Multiple Notifications in the Same Transaction

Notification Message Order

Changes to data in your organization happen in a sequential manner. However, the order in which you receive event notification messages in Streaming API 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 code shows multiple messages, one generated by the creation of a record and one generated by the update of a record.

```
{
  "channel": "/topic/InvoiceStatementUpdates",
  "clientId": "1g177wgjj14omtdo3rc10hjhm4w",
  "data": {
    "event": {
      "type": "updated",
      "createdDate": "2013-05-10T18:16:19.000+0000"
    },
    "sobject": {
      "Name": "INV-0002",
      "test_ds__Status__c": "Negotiating",
      "test_ds__Description__c": "Update to invoice statement #2",
      "Id": "a00D0000008pvxcIAA"
    }
  }
}
```

```
{
  "channel": "/topic/InvoiceStatementUpdates",
  "clientId": "1g177wgjj14omtdo3rc10hjhm4w",
  "data": {
    "event": {
      "type": "created",
      "createdDate": "2013-05-10T18:15:11.000+0000"
    },
    "sobject": {
      "Name": "INV-0003",
      "test_ds__Status__c": "Open",
      "test_ds__Description__c": "New invoice statement #1",
      "Id": "a00D0000008pvzdIAA"
    }
  }
}
```

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 and Untracked Fields**
  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. The notifications are sent regardless of whether the PushTopic tracks the other fields.

- **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 and Untracked Fields**

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. The notifications are sent regardless of whether the PushTopic tracks the other fields.

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.

```java
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 subsequently updates another field, such as `MyField__c`, a notification is generated even though this field is not tracked. The additional notification is sent because the record in question met the PushTopic criteria the first time. And because it is still the same transaction, the record remains the source of the notifications.

**Note:** In API version 36.0 and earlier, only one 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.
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.

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

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 your application exceeds these allocations, or you have scenarios for which you need to increase the number of concurrent clients, contact Salesforce.

### Reference: PushTopic

Represents a query that is the basis for notifying listeners of changes to records in an org. Available with API version 21.0 and later.

### Supported Calls

REST: DELETE, GET, PATCH, POST (query requests are specified in the URI)

SOAP: create(), delete(), describe(), describeSObjects(), query(), retrieve(), update()
Only users with the Create permission can create this record.

## Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApiVersion</td>
<td>double</td>
<td>Required. API version to use for executing the query specified in Query. It must be an API version greater than 20.0. If your query applies to a custom object from a package, this value must match the package's <code>ApiVersion</code>. Example value: 45.0</td>
</tr>
<tr>
<td>Description</td>
<td>string</td>
<td>Description of the PushTopic. Limit: 400 characters</td>
</tr>
<tr>
<td>ID</td>
<td>ID</td>
<td>System field: Globally unique string that identifies a record.</td>
</tr>
<tr>
<td>isActive</td>
<td>boolean</td>
<td>Indicates whether the record currently counts towards the organization's allocation.</td>
</tr>
<tr>
<td>IsDeleted</td>
<td>boolean</td>
<td>System field: Indicates whether the record has been moved to the Recycle Bin (<code>true</code>) or not (<code>false</code>).</td>
</tr>
<tr>
<td>Name</td>
<td>string</td>
<td>Required. Descriptive name of the PushTopic, such as MyNewCases or TeamUpdatedContacts. Limit: 25 characters. This value identifies the channel and must be unique.</td>
</tr>
<tr>
<td>NotifyForFields</td>
<td>picklist</td>
<td>Specifies which fields are evaluated to generate a notification. Valid values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• All</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Referenced (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Select</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Where</td>
</tr>
<tr>
<td>NotifyForOperations</td>
<td>picklist</td>
<td>Specifies which record events may generate a notification. Valid values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• All (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Extended</td>
</tr>
</tbody>
</table>
**PushTopic and Notifications**

The PushTopic defines when notifications are generated in the channel. This is specified by configuring the following PushTopic fields:

- **PushTopic Queries**
- **Events**
- **Notifications**

### PushTopic Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NotifyForOperationCreate</strong></td>
<td>boolean</td>
<td>true if a create operation should generate a notification, otherwise, false. Defaults to true. This field is available in API version 29.0 and later.</td>
</tr>
<tr>
<td><strong>NotifyForOperationDelete</strong></td>
<td>boolean</td>
<td>true if a delete operation should generate a notification, otherwise, false. Defaults to true. Clients must connect using the <code>cometd/29.0</code> (or later) Streaming API endpoint to receive delete and undelete event notifications. This field is available in API version 29.0 and later.</td>
</tr>
<tr>
<td><strong>NotifyForOperationUndelete</strong></td>
<td>boolean</td>
<td>true if an undelete operation should generate a notification, otherwise, false. Defaults to true. Clients must connect using the <code>cometd/29.0</code> (or later) Streaming API endpoint to receive delete and undelete event notifications. This field is available in API version 29.0 and later.</td>
</tr>
<tr>
<td><strong>NotifyForOperationUpdate</strong></td>
<td>boolean</td>
<td>true if an update operation should generate a notification, otherwise, false. Defaults to true. This field is available in API version 29.0 and later.</td>
</tr>
<tr>
<td><strong>Query</strong></td>
<td>string</td>
<td>Required. The SOQL query statement that determines which record changes trigger events to be sent to the channel. Limit: 1,300 characters</td>
</tr>
</tbody>
</table>

### Field Properties

- **Field Properties for API version 28.0 and earlier:** Create, Filter, Sort, Update
- **Field Properties for API version 29.0 and later:** Filter, Sort

In API version 29.0 and later, this field is read-only, and will not contain information about delete and undelete events. Use `NotifyForOperationCreate`, `NotifyForOperationDelete`, `NotifyForOperationUndelete` and `NotifyForOperationUpdate` to specify which record events should generate a notification.

A value of `Extended` means that neither create or update operations are set to generate events.

### Table

<table>
<thead>
<tr>
<th>Field</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NotifyForOperationCreate</strong></td>
<td>boolean</td>
<td>true if a create operation should generate a notification, otherwise, false. Defaults to true. This field is available in API version 29.0 and later.</td>
</tr>
<tr>
<td><strong>NotifyForOperationDelete</strong></td>
<td>boolean</td>
<td>true if a delete operation should generate a notification, otherwise, false. Defaults to true. Clients must connect using the <code>cometd/29.0</code> (or later) Streaming API endpoint to receive delete and undelete event notifications. This field is available in API version 29.0 and later.</td>
</tr>
<tr>
<td><strong>NotifyForOperationUndelete</strong></td>
<td>boolean</td>
<td>true if an undelete operation should generate a notification, otherwise, false. Defaults to true. Clients must connect using the <code>cometd/29.0</code> (or later) Streaming API endpoint to receive delete and undelete event notifications. This field is available in API version 29.0 and later.</td>
</tr>
<tr>
<td><strong>NotifyForOperationUpdate</strong></td>
<td>boolean</td>
<td>true if an update operation should generate a notification, otherwise, false. Defaults to true. This field is available in API version 29.0 and later.</td>
</tr>
<tr>
<td><strong>Query</strong></td>
<td>string</td>
<td>Required. The SOQL query statement that determines which record changes trigger events to be sent to the channel. Limit: 1,300 characters</td>
</tr>
</tbody>
</table>

### Field Properties

- **Field Properties:** Create, Filter, Sort, Update
CHAPTER 5

Generic Events

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 are not tied to Salesforce data changes.

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
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
  - Run a Java Client with Username and Password Login
  - Run a Java Client with OAuth Bearer Token Login
  - Generate Events Using REST

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**.

---

### 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.

```java
/*
 * Copyright (c) 2016, salesforce.com, inc.
 * All rights reserved.
 * Licensed under the BSD 3-Clause license.
 * For full license text, see LICENSE.TXT file in the repo root or
 * https://opensource.org/licenses/BSD-3-Clause
 */
package com.salesforce.emp.connector.example;

import static com.salesforce.emp.connector.LoginHelper.login;
import java.net.URL;
import java.util.Map;
import java.util.concurrent.TimeUnit;
import java.util.function.Consumer;
import com.salesforce.emp.connector.BayeuxParameters;
import com.salesforce.emp.connector.EmpConnector;
```

---

Run a Java Client with Username and Password Login
import com.salesforce.emp.connector.LoginHelper;
import com.salesforce.emp.connector.TopicSubscription;

/**
 * An example of using the EMP connector using login credentials
 */
public class LoginExample {
  public static void main(String[] argv) throws Exception {
    if (argv.length < 3 || argv.length > 4) {
      System.err.println(
          "Usage: LoginExample username password channel [replayFrom]");
      System.exit(1);
    }
    long replayFrom = EmpConnector.REPLAY_FROM_EARLIEST;
    if (argv.length == 4) {
      replayFrom = Long.parseLong(argv[3]);
    }

    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();
    Consumer<Map<String, Object>> consumer = event -> System.out.println(String.format("Received:
%s", event));
    EmpConnector connector = new EmpConnector(params);
    connector.setBearerTokenProvider(tokenProvider);
    connector.start().get(5, TimeUnit.SECONDS);
    TopicSubscription subscription = connector.subscribe(argv[2], replayFrom, consumer).get(5, TimeUnit.SECONDS);
    System.out.println(String.format("Subscribed: %s", subscription));
  }
}

3. Run the LoginExample class and provide arguments.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>Username of the logged-in user.</td>
</tr>
<tr>
<td>password</td>
<td>Password for the username (or logged-in user).</td>
</tr>
<tr>
<td>channel</td>
<td>/u/notifications/ExampleUserChannel</td>
</tr>
</tbody>
</table>
The sample fetches the earliest saved events within the past 24 hours. 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.
- 2—Get all new events sent after subscription and all past events within the past 24 hours.
- 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:

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

Generally, do not 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.

**Prerequisites**

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

See Set Up Authentication with OAuth 2.0. Also see Authenticate Apps with OAuth in Salesforce Help and Understanding Authentication in the REST API Developer Guide.

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.

```java
package com.salesforce.emp.connector.example;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;
import java.util.concurrent.TimeUnit;
import java.util.function.Consumer;
import com.salesforce.emp.connector.BayeuxParameters;
import com.salesforce.emp.connector.EmpConnector;
import com.salesforce.emp.connector.TopicSubscription;
import org.cometd.bayeux.Channel;

/**<code>
* Copyright (c) 2016, salesforce.com, inc. All rights reserved. Licensed under the BSD 3-Clause license. For full license text, see LICENSE.TXT file in the repo root or https://opensource.org/licenses/BSD-3-Clause
*/
package com.salesforce.emp.connector.example;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.Map;
import java.util.concurrent.TimeUnit;
import java.util.function.Consumer;
import com.salesforce.emp.connector.BayeuxParameters;
import com.salesforce.emp.connector.EmpConnector;
import com.salesforce.emp.connector.TopicSubscription;
import org.cometd.bayeux.Channel;

/**<code>
*/
```
public class BearerTokenExample {
    public static void main(String[] argv) throws Exception {
        if (argv.length < 2 || argv.length > 4) {
            System.err.println("Usage: BearerTokenExample url token channel [replayFrom]");
            System.exit(1);
        }
        long replayFrom = EmpConnector.REPLAY_FROM_EARLIEST;
        if (argv.length == 4) {
            replayFrom = Long.parseLong(argv[3]);
        }
        BayeuxParameters params = new BayeuxParameters() {
            @Override
            public String bearerToken() {
                return argv[1];
            }
            @Override
            public URL host() {
                try {
                    return new URL(argv[0]);
                } catch (MalformedURLException e) {
                    throw new IllegalArgumentException(String.format("Unable to create url: %s", argv[0]), e);
                }
            }
        };
        Consumer<Map<String, Object>> consumer = event -> System.out.println(
            String.format("Received:
%s", event));
        EmpConnector connector = new EmpConnector(params);
        connector.addListener(Channel.META_CONNECT, new LoggingListener(true, true))
            .addListener(Channel.META_DISCONNECT, new LoggingListener(true, true))
            .addListener(Channel.META_HANDSHAKE, new LoggingListener(true, true));
        connector.start().get(5, TimeUnit.SECONDS);
        TopicSubscription subscription = connector.subscribe(
            argv[2], replayFrom, consumer).get(5, TimeUnit.SECONDS);
        System.out.println(String.format("Subscribed: %s", subscription));
    }
}

3. Run the BearerTokenExample class, and provide the following argument values.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>Username of the logged-in user.</td>
</tr>
<tr>
<td>Argument</td>
<td>Value</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>password</td>
<td>Password for the username (or logged-in user).</td>
</tr>
<tr>
<td>channel</td>
<td>/u/notifications/ExampleUserChannel</td>
</tr>
</tbody>
</table>

The sample fetches the earliest saved events within the past 24 hours. 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.
- –2—Get all new events sent after subscription and all past events within the past 24 hours.
- 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:-2]
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.

You’ll use Workbench to access REST API and send notifications. Workbench is a free, open source, community-supported tool (see the Help page in Workbench). Salesforce provides a hosted instance of Workbench for demonstration purposes only—Salesforce recommends that you do not use this hosted instance of Workbench to access data in a production database. If you want to use Workbench for your production database, you can download, host, and configure it using your own resources. You can download Workbench from https://github.com/forceworkbench/forceworkbench.

1. In a browser, navigate to https://workbench.developerforce.com/login.php.
2. For Environment, select Production.
3. For API Version, select 45.0.
4. Accept the terms of service and click Login with Salesforce.
5. Once you successfully establish a connection to your database, you land on the Select page.
6. Find the StreamingChannel ID by clicking queries > SOQL Query and doing a SOQL query for SELECT Name, ID FROM StreamingChannel. Copy down the StreamingChannel ID for /u/notifications/ExampleUserChannel.
7. Click utilities > REST Explorer.
8. In the URL field, enter /services/data/v<API version>/sobjects/StreamingChannel/<Streaming Channel ID>/push, where v<API version> is v45.0 and <Streaming Channel ID> is the ID of the StreamingChannel you found in Step 6.
9. Set the HTTP method by selecting POST. In Request Body, enter the JSON request body shown in “Example POST REST request body” below.
10. With your Java subscriber client running, click **Execute**. This sends the event to all subscribers on the channel. You should receive the notification with the payload text in your Java client. The REST method response will indicate the number of subscribers the event was sent to (in this case, –1, because the event was set to broadcast to all subscribers).

You’ve successfully sent a notification to a subscriber using generic streaming. Note that 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.

**Example**: Example POST REST request body:

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

**Generic Streaming Allocations**

These default allocations apply to consumers of generic streaming.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum streaming channels per org</td>
<td>1,000</td>
<td>1,000</td>
<td>1,000</td>
<td>200</td>
</tr>
<tr>
<td>Maximum number of concurrent clients (subscribers) across all channels and for all event types</td>
<td>2,000</td>
<td>1,000</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>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)</td>
<td>1,000,000</td>
<td>200,000</td>
<td>100,000</td>
<td>10,000</td>
</tr>
<tr>
<td>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)</td>
<td>100,000</td>
<td>100,000</td>
<td>100,000</td>
<td>10,000</td>
</tr>
</tbody>
</table>

**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 your application exceeds these allocations, or you have scenarios for which you need to increase the number of concurrent clients, contact Salesforce.
Reference: StreamingChannel

Represents a channel that is the basis for notifying listeners of generic Streaming API events. Available with API version 29.0 and later.

Supported Calls

REST: DELETE, GET, PATCH, POST (query requests are specified in the URI)

SOAP: create(), delete(), describe(), describeLayout(), describeSObjects(), getDeleted(), getUpdated(), query(), retrieve(), undelete(), update()

Special Access Rules

- This object is available only if Streaming API is enabled for your org.
- Only users with the Create permission can create this record.
- You can create a permission set and grant users read and create access to all streaming channels in the org. This access isn’t for a specific channel, like with user sharing.
- You can apply user sharing to StreamingChannel. You can restrict access to receiving or sending events on a channel by sharing channels with specific users or groups. 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.

Dynamic Streaming Channel

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.

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| Description | string     | Description of the StreamingChannel. Limit: 255 characters. **Field Properties:** Create, Filter, Group, Nillable, Sort, Update  
**Label:** Description |
<p>| ID          | ID         | System field: Globally unique string that identifies a StreamingChannel record. <strong>Field Properties:</strong> Default on create, Filter, Group, idLookup, Sort |
| IsDeleted   | boolean    | System field: Indicates whether the record has been moved to the Recycle Bin (true) or not (false). <strong>Field Properties:</strong> Default on create, Filter, Group, Sort |</p>
<table>
<thead>
<tr>
<th>Field</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsDynamic</td>
<td>boolean</td>
<td>true if the channel gets dynamically created on subscribe if necessary, false otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Field Properties:</strong> Default on create, Filter, Group, Sort</td>
</tr>
<tr>
<td>LastReferencedDate</td>
<td>date</td>
<td>The timestamp for when the current user last viewed a record related to this record.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Field Properties:</strong> Filter, Sort</td>
</tr>
<tr>
<td>LastViewedDate</td>
<td>date</td>
<td>The timestamp for when the current user last viewed this record. If this value is null, this record might only have been referenced (LastReferencedDate) and not viewed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Field Properties:</strong> Filter, Sort</td>
</tr>
<tr>
<td>Name</td>
<td>string</td>
<td>Required. Descriptive name of the StreamingChannel. Limit: 80 characters, alphanumeric and &quot;_&quot;, &quot;/&quot; characters only. Must start with &quot;/u/&quot;. This value identifies the channel and must be unique.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Field Properties:</strong> Create, Filter, Group, idLookup, Sort, Update</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Label:</strong> Streaming Channel Name</td>
</tr>
<tr>
<td>OwnerId</td>
<td>reference</td>
<td>The ID of the owner of the StreamingChannel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Field Properties:</strong> Create, Default on create, Filter, Group, Sort, Update</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Label:</strong> Owner Name</td>
</tr>
</tbody>
</table>

Reference: Streaming Channel Push REST API

Gets subscriber information, and pushes notifications for streaming channels.

Syntax

**URI**

/vXX.X/sobjects/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:
Each push event payload contains the following fields:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>payload</td>
<td>string</td>
<td>Information sent with notification. Cannot exceed 3,000 single-byte characters.</td>
</tr>
<tr>
<td>userIds</td>
<td>array of User IDs</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Request parameters**

None

**Response data**

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnlineUserIds</td>
<td>array of User IDs</td>
<td>User IDs of currently subscribed users to this channel.</td>
</tr>
<tr>
<td>ChannelName</td>
<td>string</td>
<td>Name of the channel, for example, /u/notifications/ExampleUserChannel.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fanoutCount</td>
<td>number</td>
<td>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, fanoutCount will be –1. If no active subscribers were found for the channel, fanoutCount will be 0.</td>
</tr>
<tr>
<td>userOnlineStatus</td>
<td>array of User online status information</td>
<td>List of User IDs the notification was sent to and their listener status. If true the User ID is actively subscribed and listening, otherwise false.</td>
</tr>
</tbody>
</table>

**Example**

The following is an example JSON response of a GET request for services/data/v29.0/objects/StreamingChannel/0M6D000000000g7KXA/push:

```json
{
  "OnlineUserIds" : [ "005D0000001X11XAG" ],
  "ChannelName" : "/u/notifications/ExampleUserChannel"
}
```
Using a POST request to services/data/v29.0/sobjects/StreamingChannel/0M6D000000000g7KXA/push with a request JSON body of:

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

the JSON response data looks something like:

```json
[
    {
        "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:

<table>
<thead>
<tr>
<th>Limit Label</th>
<th>Description</th>
<th>API Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>DailyDurableGenericStreamingApiEvents</td>
<td>Maximum and remaining number of generic events in the past 24 hours for Durable Streaming</td>
<td>37.0 and later</td>
</tr>
<tr>
<td>DailyDurableStreamingApiEvents</td>
<td>Maximum and remaining number of PushTopic events in the past 24 hours for Durable Streaming</td>
<td>37.0 and later</td>
</tr>
<tr>
<td>DurableStreamingApiConcurrentClients</td>
<td>Maximum and remaining number of concurrent clients (subscribers) for Durable Streaming</td>
<td>37.0 and later</td>
</tr>
<tr>
<td>DailyGenericStreamingApiEvents</td>
<td>Maximum and remaining number of generic events in the past 24 hours</td>
<td>36.0 and earlier</td>
</tr>
<tr>
<td>DailyStreamingApiEvents</td>
<td>Maximum and remaining number of PushTopic events in the past 24 hours</td>
<td>36.0 and earlier</td>
</tr>
<tr>
<td>StreamingApiConcurrentClients</td>
<td>Maximum and remaining number of concurrent clients (subscribers)</td>
<td>36.0 and earlier</td>
</tr>
</tbody>
</table>

REST API Endpoint

/vXX.X/limits/

For more information, see Limits and List Organization Limits in the REST API Developer Guide.
Index

S
Security 59
Streaming API
  Getting started 1
Streaming Channel Push REST Resource 85
Streaming Events
  feature comparison 12
Subscriptions
  filtered 69

T
Terms 2
Timeouts 50

U
Using Streaming API 49

W
Which Event Do I Use? 12