Version 30.0: Spring '14

Database.com Streaming API Developer's Guide

Last updated: May 5, 2014

Table of Contents

GETTING STARTED WITH STREAMING API

Chapter 1

Introducing Streaming API

In this chapter ...

Use Streaming API to receive notifications for changes to Database.com data that match a SOQL query you define, in a secure and scalable way.

These events can be received by:

- Application servers outside of Database.com.
- Clients outside the Database.com application.

The sequence of events when using Streaming API is as follows:

1. Create a PushTopic based on a SOQL query. This 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.

Streaming API is useful when you want notifications to be pushed from the server to the client based on criteria that you define. Consider the following applications for Streaming API:

Applications that poll frequently

Applications that have constant polling action against the Database.com infrastructure, consuming unnecessary API calls and processing time, would benefit from Streaming API which 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 organization. This enables you to reduce the number of API calls and improve performance.

Note: You can use Streaming API with any organization as long as you enable the API. This includes both Salesforce and Database.com organizations.

Push Technology Overview

Push technology is a model of Internet-based communication in which information transfer is initiated from a server to the client. Also called the publish/subscribe model, 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 that's sent by the server is typically specified in advance. When using Streaming API, you specify the information the client receives by creating a PushTopic. The client then subscribes to the PushTopic channel to be notified of events that match the PushTopic criteria.

In push technology, the server pushes out information to the client after the client has subscribed to a channel of information. In order for the client to receive the information, the client must maintain a connection to the server. 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

The Bayeux protocol and CometD both use 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. The Database.com servers use version 2.0 of CometD.
- 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 following resources:

- *CometD documentation: http://cometd.org/documentation*
- *Bayeux protocol documentation: http://cometd.org/documentation/bayeux*
- *Bayeux protocol specification: http://cometd.org/documentation/bayeux/spec*

Streaming API supports the following CometD methods:

Method	Description
`connect`	The client connects to the server.
`disconnect`	The client disconnects from the server.
`handshake`	The client performs a handshake with the server and establishes a long polling connection.
`subscribe`	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.
`unsubscribe`	The client unsubscribes from a channel.

Streaming API Terms

The following table lists terms related to Streaming API.

Term	Description
Event	The creation, update, delete, or undelete of a record. Each event may 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 record that you create. The essential element of a PushTopic is the SOQL query. The PushTopic defines a Streaming API channel.

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 the Streaming API in three stages:

1. Sends a handshake request.
2. Sends a subscription request to a channel.
3. Connects using *long polling*.

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.

The client receives events from the server while it maintains a long-lived connection.

- If the client receives events, it should reconnect immediately 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.
- If no events are generated and the client is waiting and the server closes the connection, after two minutes the client should reconnect immediately.

If a long-lived connection is lost due to unexpected network disruption, CometD will automatically attempt to reconnect. If this reconnection is successful, clients must re-subscribe, since this new connection has gone through a re-handshake that removes previous subscribers. Clients can listen to the `meta/handshake` meta channel to receive notifications when a connection is lost and re-established.

For details about these steps, see Bayeux Protocol, CometD, and Long Polling.

Message Reliability

Streaming API doesn't guarantee durability and reliable delivery of notifications. Streaming servers don't maintain any client state and don't keep track of what's delivered. The client may not receive messages for a variety of reasons, including:

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

Chapter 2

Quick Start Using Workbench

This quick start shows you how to get started with Streaming API by using Workbench. This quick start takes you step-by-step through the process of using Streaming API to receive a notification when a record is updated.

Prerequisites

You need access and appropriate permissions to complete the quick start steps.

- Access to a Developer Edition organization.

  If you are not already a member of the Force.com developer community, go to `http://developer.force.com/join` 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, click **Customize** > **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

The first step is to create an InvoiceStatement object. 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. Click **Create** > **Objects**.
2. Click **New Custom Object** and fill in the custom object definition.

- In the **Label field**, type `Invoice Statement`.
- In the **Plural Label** field, type `Invoice Statements`.
- Select **Starts with vowel sound**.
- In the **Record Name** field , type `Invoice Number`.
- In the **Data Type** field , select `Auto Number`.
- In the **Display Format** field, type `INV-{0000}`.
- In the **Starting Number** field, type `1`.

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

```
Open
Closed
Negotiating
Pending
```

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

   Your 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. Alternatively, you can also use Workbench to create a PushTopic.

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

```
PushTopic pushTopic = new PushTopic();
pushTopic.Name = 'InvoiceStatementUpdates';
pushTopic.Query = 'SELECT Id, Name, Status__c, Description__c FROM Invoice_Statement__c';
pushTopic.ApiVersion = 30.0;
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: Subscribe to the PushTopic Channel

In this step, you'll subscribe to the channel you created with the PushTopic record in the previous step.

Important: Workbench is a free, open source, community-supported tool (see the Help page in Workbench). Salesforce.com provides a hosted instance of Workbench for demonstration purposes only—salesforce.com 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 *http://code.google.com/p/forceworkbench/downloads/list*.

1. In your browser, navigate to `http://workbench.developerforce.com`.
2. For Environment, select **Production**.
3. For API Version, select 30.0.
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. Click **queries** > **Streaming Push Topics**.
7. In the Push Topic field, select **InvoiceStatementUpdates**.
8. Click **Subscribe**.

You'll see the connection and response information and a response like "Subscribed to /topic/InvoiceStatementUpdates."

Keep this browser window open and make sure the connection doesn't time out. You'll be able to see the event notifications triggered by the InvoiceStatement record you create in the next step.

Step 4: Test the PushTopic Channel

Make sure the browser that you used in Step 3: Subscribe to the PushTopic Channel stays open and the connection doesn't time out. You'll view event notifications in this browser.

The final step is to test the PushTopic channel by creating a new InvoiceStatement record in Workbench and viewing the event notification.

1. In a new browser window, open an instance of Workbench and log in using the same username by following the steps in Step 3: Subscribe to the PushTopic Channel.

   Note: If the user that makes an update to a record and the user that's subscribed to the channel don't share records, then the subscribed user won't receive the notification. For example, if the sharing model for the organization is private.

2. Click **data** > **Insert**.
3. For Object Type, select **Invoice_Statement__c**. Ensure that the **Single Record** field is selected, and click **Next**.
4. Type in a value for the **Description__c** field.
5. Click **Confirm Insert**.
6. Switch over to your Streaming Push Topics browser window. You'll see a notification that the invoice statement was created. The notification returns the `Id`, `Status__c`, and `Description__c` fields that you defined in the SELECT statement of your PushTopic query. The message looks something like this:

```
{
  "channel": "/topic/InvoiceStatementUpdates",
  "data": {
    "event": {
      "type": "created",
      "createdDate": "2011-11-14T17:33:45.000+0000"
    },
    "sobject": {
      "Name": "INV-0004",
      "Id": "a00D0000008oLi8IAE",
      "Description__c": "Test invoice statement",
      "Status__c": "Open"
    }
  }
}
```

CODE EXAMPLES

Chapter 3

Example: Java Client

This code example shows you how to implement Streaming API from a Java client. When you run the Java client, it subscribes to the channel and receives notifications.

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 Force.com developer community, go to `http://developer.force.com/join` 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, click **Customize** > **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

To perform this example, you must first create the InvoiceStatement object. If you haven't already created this object, see Step 1: Create an Object.

Step 2: Create a PushTopic

To perform this example, you must create a PushTopic. If you haven't already done so, see Step 2: Create a PushTopic.

Step 3: Download the JAR Files

Add the following library files to the build path of your Java client application for Streaming API.

1. Download the compressed archive file from *http://download.cometd.org/cometd-2.3.1-distribution.tar.gz*.
2. Extract the following JAR files from `cometd-2.3.1.tgz`:
   - `cometd-2.3.1/cometd-java/bayeux-api/target/bayeux-api-2.3.1.jar`
   - `cometd-2.3.1/cometd-java/cometd-java-client/target/cometd-java-client-2.3.1.jar`
   - `cometd-2.3.1/cometd-java/cometd-java-common/target/cometd-java-common-2.3.1.jar`
3. Download the compressed archive file from the following URL:
   http://dist.codehaus.org/jetty/jetty-hightide-7.4.4/jetty-hightide-7.4.4.v20110707.tar.gz.
   Jetty Hightide is a distribution of the Jetty open source Web container. For more information, see the Jetty Hightide documentation.
4. Extract the following JAR files from `jetty-hightide-7.4.4.v20110707.tar.gz`.
   - `jetty-hightide-7.4.4.v20110707/lib/jetty-client-7.4.4.v20110707.jar`
   - `jetty-hightide-7.4.4.v20110707/lib/jetty-http-7.4.4.v20110707.jar`
   - `jetty-hightide-7.4.4.v20110707/lib/jetty-io-7.4.4.v20110707.jar`
   - `jetty-hightide-7.4.4.v20110707/lib/jetty-util-7.4.4.v20110707.jar`

Step 4: Add the Source Code

1. Add the following code to a Java source file named `StreamingClientExample.java`. This code subscribes to the PushTopic channel and handles the streaming information.

```
package demo;

import org.cometd.bayeux.Channel;
import org.cometd.bayeux.Message;
import org.cometd.bayeux.client.ClientSessionChannel;
import org.cometd.bayeux.client.ClientSessionChannel.MessageListener;
import org.cometd.client.BayeuxClient;
import org.cometd.client.transport.ClientTransport;
import org.cometd.client.transport.LongPollingTransport;

import org.eclipse.jetty.client.ContentExchange;
import org.eclipse.jetty.client.HttpClient;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
```

```
/**
 * This example demonstrates how a streaming client works
 * against the Salesforce Streaming API.
 **/

public class StreamingClientExample {

    // This URL is used only for logging in. The LoginResult
    // returns a serverUrl which is then used for constructing
    // the streaming URL. The serverUrl points to the endpoint
    // where your organization is hosted.

    static final String LOGIN_ENDPOINT = "https://login.database.com";
   private static final String USER_NAME = "change_this_to_your_testuser@yourcompany.com";

    private static final String PASSWORD = "change_this_to_your_testpassword";
    // NOTE: Putting passwords in code is not a good practice and not recommended.

    // Set this to true only when using this client
    // against the Summer'11 release (API version=22.0).
    private static final boolean VERSION_22 = false;
    private static final boolean USE_COOKIES = VERSION_22;

    // The channel to subscribe to. Same as the name of the PushTopic.
    // Be sure to create this topic before running this sample.
    private static final String CHANNEL = VERSION_22 ? "/InvoiceStatementUpdates" :
"/topic/InvoiceStatementUpdates";
    private static final String STREAMING_ENDPOINT_URI = VERSION_22 ?
        "/cometd" : "/cometd/30.0";

    // The long poll duration.
    private static final int CONNECTION_TIMEOUT = 20 * 1000;  // milliseconds
    private static final int READ_TIMEOUT = 120 * 1000; // milliseconds

    public static void main(String[] args) throws Exception {

     System.out.println("Running streaming client example....");

        final BayeuxClient client = makeClient();
        client.getChannel(Channel.META_HANDSHAKE).addListener
            (new ClientSessionChannel.MessageListener() {

            public void onMessage(ClientSessionChannel channel, Message message) {

             System.out.println("[CHANNEL:META_HANDSHAKE]: " + message);

                boolean success = message.isSuccessful();
                if (!success) {
                    String error = (String) message.get("error");
                    if (error != null) {
                        System.out.println("Error during HANDSHAKE: " + error);
                        System.out.println("Exiting...");
                        System.exit(1);
                    }

                    Exception exception = (Exception) message.get("exception");
                    if (exception != null) {
                        System.out.println("Exception during HANDSHAKE: ");
                        exception.printStackTrace();
                        System.out.println("Exiting...");
                        System.exit(1);

                    }
                }
            }

        });

        client.getChannel(Channel.META_CONNECT).addListener(
```

```
            new ClientSessionChannel.MessageListener() {
            public void onMessage(ClientSessionChannel channel, Message message) {

                System.out.println("[CHANNEL:META_CONNECT]: " + message);

                boolean success = message.isSuccessful();
                if (!success) {
                    String error = (String) message.get("error");
                    if (error != null) {
                        System.out.println("Error during CONNECT: " + error);
                        System.out.println("Exiting...");
                        System.exit(1);
                    }
                }
            }

        });

        client.getChannel(Channel.META_SUBSCRIBE).addListener(
            new ClientSessionChannel.MessageListener() {

            public void onMessage(ClientSessionChannel channel, Message message) {

             System.out.println("[CHANNEL:META_SUBSCRIBE]: " + message);
                boolean success = message.isSuccessful();
                if (!success) {
                    String error = (String) message.get("error");
                    if (error != null) {
                        System.out.println("Error during SUBSCRIBE: " + error);
                        System.out.println("Exiting...");
                        System.exit(1);
                    }
                }
            }
        });



        client.handshake();
        System.out.println("Waiting for handshake");

        boolean handshaken = client.waitFor(10 * 1000, BayeuxClient.State.CONNECTED);
        if (!handshaken) {
            System.out.println("Failed to handshake: " + client);
            System.exit(1);
        }



        System.out.println("Subscribing for channel: " + CHANNEL);

        client.getChannel(CHANNEL).subscribe(new MessageListener() {
            @Override
            public void onMessage(ClientSessionChannel channel, Message message) {
                System.out.println("Received Message: " + message);
            }
        });

        System.out.println("Waiting for streamed data from your organization ...");
        while (true) {
            // This infinite loop is for demo only,
            // to receive streamed events on the
            // specified topic from your organization.
        }
    }



    private static BayeuxClient makeClient() throws Exception {
        HttpClient httpClient = new HttpClient();
```

```
        httpClient.setConnectTimeout(CONNECTION_TIMEOUT);
        httpClient.setTimeout(READ_TIMEOUT);
        httpClient.start();

        String[] pair = SoapLoginUtil.login(httpClient, USER_NAME, PASSWORD);

        if (pair == null) {
            System.exit(1);
        }

        assert pair.length == 2;
        final String sessionid = pair[0];
        String endpoint = pair[1];
        System.out.println("Login successful!\nEndpoint: " + endpoint
            + "\nSessionid=" + sessionid);

        Map<String, Object> options = new HashMap<String, Object>();
        options.put(ClientTransport.TIMEOUT_OPTION, READ_TIMEOUT);
        LongPollingTransport transport = new LongPollingTransport(
          options, httpClient) {

         @Override
            protected void customize(ContentExchange exchange) {
                super.customize(exchange);
                exchange.addRequestHeader("Authorization", "OAuth " + sessionid);
            }
        };

        BayeuxClient client = new BayeuxClient(salesforceStreamingEndpoint(
            endpoint), transport);
        if (USE_COOKIES) establishCookies(client, USER_NAME, sessionid);
        return client;
    }

    private static String salesforceStreamingEndpoint(String endpoint)
        throws MalformedURLException {
        return new URL(endpoint + STREAMING_ENDPOINT_URI).toExternalForm();
    }

    private static void establishCookies(BayeuxClient client, String user,
        String sid) {
        client.setCookie("com.salesforce.LocaleInfo", "us", 24 * 60 * 60 * 1000);
        client.setCookie("login", user, 24 * 60 * 60 * 1000);
        client.setCookie("sid", sid, 24 * 60 * 60 * 1000);
        client.setCookie("language", "en_US", 24 * 60 * 60 * 1000);
    }
}
```

2. Edit `StreamingClientExample.java` and modify the following values:

File Name	Static Resource Name
USER_NAME	Username of the logged-in user
PASSWORD	Password for the *USER_NAME* (or logged-in user)
CHANNEL	`/topic/InvoiceStatementUpdates`
LOGIN_ENDPOINT	`https://test.database.com` (Only if you are using a test database. If you are in a production organization, no change is required for LOGIN_ENDPOINT.)

3. Add the following code to a Java source file named `SoapLoginUtil.java`. This code sends a username and password to the server and receives the session ID.

Important: Never handle the usernames and passwords of others. Before using in a production environment, delegate the login to OAuth.

```
package demo;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.MalformedURLException;
import java.net.URL;

import org.eclipse.jetty.client.ContentExchange;
import org.eclipse.jetty.client.HttpClient;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

public final class SoapLoginUtil {

    // The enterprise SOAP API endpoint used for the login call in this example.
   private static final String SERVICES_SOAP_PARTNER_ENDPOINT = "/services/Soap/u/22.0/";


    private static final String ENV_START =
            "<soapenv:Envelope xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope/'
 "
                    + "xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' " +
                    "xmlns:urn='urn:partner.soap.sforce.com'><soapenv:Body>";

    private static final String ENV_END = "</soapenv:Body></soapenv:Envelope>";

    private static byte[] soapXmlForLogin(String username, String password)
        throws UnsupportedEncodingException {
        return (ENV_START +
                "  <urn:login>" +
                "    <urn:username>" + username + "</urn:username>" +
                "    <urn:password>" + password + "</urn:password>" +
                "  </urn:login>" +
                ENV_END).getBytes("UTF-8");
    }

    public static String[] login(HttpClient client, String username, String password)
            throws IOException, InterruptedException, SAXException,
                ParserConfigurationException {

        ContentExchange exchange = new ContentExchange();
        exchange.setMethod("POST");
        exchange.setURL(getSoapURL());
        exchange.setRequestContentSource(new ByteArrayInputStream(soapXmlForLogin(
            username, password)));
        exchange.setRequestHeader("Content-Type", "text/xml");
        exchange.setRequestHeader("SOAPAction", "''");
        exchange.setRequestHeader("PrettyPrint", "Yes");

        client.send(exchange);
        exchange.waitForDone();
        String response = exchange.getResponseContent();

        SAXParserFactory spf = SAXParserFactory.newInstance();
        spf.setNamespaceAware(true);
        SAXParser saxParser = spf.newSAXParser();

        LoginResponseParser parser = new LoginResponseParser();
```

```
        saxParser.parse(new ByteArrayInputStream(
            response.getBytes("UTF-8")), parser);

        if (parser.sessionId == null || parser.serverUrl == null) {
            System.out.println("Login Failed!\n" + response);
            return null;
        }

        URL soapEndpoint = new URL(parser.serverUrl);
        StringBuilder endpoint = new StringBuilder()
            .append(soapEndpoint.getProtocol())
            .append("://")
            .append(soapEndpoint.getHost());

        if (soapEndpoint.getPort() > 0) endpoint.append(":")
            .append(soapEndpoint.getPort());
        return new String[] {parser.sessionId, endpoint.toString()};
    }

    private static String getSoapURL() throws MalformedURLException {
        return new URL(StreamingClientExample.LOGIN_ENDPOINT +
            getSoapUri()).toExternalForm();
    }

    private static String getSoapUri() {
        return SERVICES_SOAP_PARTNER_ENDPOINT;
    }

    private static class LoginResponseParser extends DefaultHandler {

        private boolean inSessionId;
        private String sessionId;

        private boolean inServerUrl;
        private String serverUrl;

        @Override
        public void characters(char[] ch, int start, int length) {
            if (inSessionId) sessionId = new String(ch, start, length);
            if (inServerUrl) serverUrl = new String(ch, start, length);
        }

        @Override
        public void endElement(String uri, String localName, String qName) {
            if (localName != null) {
                if (localName.equals("sessionId")) {
                    inSessionId = false;
                }

                if (localName.equals("serverUrl")) {
                    inServerUrl = false;
                }
            }
        }

        @Override
        public void startElement(String uri, String localName,
            String qName, Attributes attributes) {
            if (localName != null) {
                if (localName.equals("sessionId")) {
                    inSessionId = true;
                }

                if (localName.equals("serverUrl")) {
                    inServerUrl = true;
                }
            }
        }
    }
}
```

4. In a different browser window, create or modify an InvoiceStatement. After you create or change data that corresponds to the query in your PushTopic, the output looks something like this:

```
Running streaming client example....
Login successful!
Endpoint: https:www.database.com
Sessionid=00DD0000000FSp9!AQIAQIVjGYijFhiAROTc455T6kEVeJGXuW5VCnp
    LANCMawS7.p5fXbjYlqCgx7They_zFjmP5n9HxvfUA6xGSGtC1Nb6P4S.

Waiting for handshake
[CHANNEL:META_HANDSHAKE]:
{
    "id":"1",
    "minimumVersion":"1.0",
    "supportedConnectionTypes":["long-polling"],
    "successful":true,
    "channel":"/meta/handshake",
    "clientId":"31t0cjzfbgnfqn1rggumba0k98u",
    "version":"1.0"
}

[CHANNEL:META_CONNECT]:
{
    "id":"2",
    "successful":true,
    "advice":{"interval":0,"reconnect":"retry","timeout":110000},
    "channel":"/meta/connect"}
    Subscribing for channel: /topic/InvoiceStatementUpdates
    Waiting for streamed data from your organization ...
[CHANNEL:META_SUBSCRIBE]:
{
    "id":"4",
    "subscription":"/topic/InvoiceStatementUpdates",
    "successful":true,
    "channel":"/meta/subscribe"
}

[CHANNEL:META_CONNECT]:
{
    "id":"3",
    "successful":true,
    "channel":"/meta/connect"
}

Received Message:
{
"data":
    {
      "sobject":
      {
        "Name":"INV-0002",
        "Id":"001D000000J3fTHIAZ",
        "Status__c":"Pending"},
        "event":{"type":"updated",
        "createdDate":"2011-09-06T18:51:08.000+0000"
      }
    },
    "channel":"/topic/InvoiceStatementUpdates"
}

[CHANNEL:META_CONNECT]:
{
    "id":"5",
    "successful":true,
    "channel":"/meta/connect"
}
```

Chapter 4

Examples: Authentication

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

Setting Up Authentication for Developer Testing

To set up authorization for developer testing:

Important: This authorization method should only be used for testing and never in a production environment.

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 Database.com server.

For more details, see Step 4: Add the Source Code.

Setting 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 Database.com REST API Developer's 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 Database.com, from Setup, click **Create** > **Apps**. 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." Database.com 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 *http://wiki.developerforce.com/page/Digging_Deeper_into_OAuth_2.0_on_Force.com*.

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

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

```
HttpClient httpclient = new DefaultHttpClient();
HttpPost post = new HttpPost(baseURL);

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

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

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

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

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

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

    // Execute the request.
    System.out.println("POST " + baseUrl + "...\n");
    HttpResponse response = client.execute(oauthPost);
    int code = response.getStatusLine().getStatusCode();
```

```
    Map<String, String> oauthLoginResponse = (Map<String, String>)
        JSON.parse(EntityUtils.toString(response.getEntity()));
    System.out.println("OAuth login response");
    for (Map.Entry<String, String> entry : oauthLoginResponse.entrySet())
    {
        System.out.println(String.format("  %s = %s", entry.getKey(), entry.getValue()));
    }
    System.out.println("");

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

    // Use the user info in interesting ways.
    System.out.println("Username is " + userInfo.get("username"));
    System.out.println("User's email is " + userInfo.get("email"));
    Map<String, String> urls = (Map<String, String>)userInfo.get("urls");
    System.out.println("REST API url is " + urls.get("rest").replace("{version}", "30.0"));
}
```

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 =
00D30000000ehjI!ARYAQHc.0Mlmz.DCg3HRNF.SmsSn5njPkry2SM6pb6rjCOqfAODaUkv5CGksRSPRb.xb
  signature = 8M9VWBoaEk+Bs//yD+BfrUR/+5tkNLgXAIwal1PMwsY=

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://c.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/v30.0/
```

Chapter 5

Working with PushTopics

In this chapter ...

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.

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

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:

```
{
  "channel": "/topic/InvoiceStatementUpdates",
  "data":
  {
    "event":
    {
      "type": "updated",
      "createdDate": "2011-11-03T15:59:06.000+0000"
    },
    "sobject":
    {
      "Name": "INV-0001",
      "Id": "a00D0000008o6y8IAA",
      "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
- 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:

```
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.

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.

- Query on the Merchandise__c custom object

```
SELECT Id, Name FROM Merchandise__c
```

- Query on the Merchandise__c custom object using the `LIKE` operator

```
SELECT Id, Name, Price__c FROM Merchandise__c WHERE Name LIKE 'Pencil%'
```

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
- Compound address or geolocation fields
- `NOT`
  - Example query: `SELECT Id FROM Account WHERE NOT Name = 'Salesforce.com'`
  - Error message: `INVALID_FIELD, 'NOT' is not supported`

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

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

- `OFFSET`
  - Example query: `SELECT Id, Name FROM Account WHERE City = 'New York' OFFSET 10`
  - Error message: `INVALID_FIELD, 'OFFSET' clause is not allowed`
- `TYPEOF`
  - Example query: `SELECT TYPEOF Owner WHEN User THEN LastName ELSE Name END FROM Case`
  - Error message: `INVALID_FIELD, 'TYPEOF' clause is not allowed`

  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.com.

Event Notification Rules

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

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

Events

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

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

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	Description
`All` (default)	Evaluate a record to possibly generate a notification whether the record has been created or updated.
`Create`	Evaluate a record to possibly generate a notification only if the record has been created.
`Update`	Evaluate a record to possibly generate a notification only if the record has been updated.
`Extended`	A value of `Extended` 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 Database.com organizations configured to generate delete and undelete notifications.

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

Notifications

After a record is created or updated (an event), the record is evaluated against the PushTopic query and a notification may 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:

`NotifyForFields` Value	Description
`All`	Notifications are generated for all record field changes, provided the values of the fields referenced in the WHERE clause match the values specified in the WHERE clause.
`Referenced (default)`	Changes to fields referenced in both the SELECT clause and WHERE clause are evaluated. Notifications are generated for all records where a field referenced in the SELECT clause changes or a field referenced in the WHERE clause changes and the values of the fields referenced in the WHERE clause match the values specified in the WHERE clause.
`Select`	Changes to fields referenced in the SELECT clause are evaluated. Notifications are generated for all records where a field referenced in the SELECT clause changes and the values of the fields referenced in the WHERE clause match the values specified in the WHERE clause.

NotifyForFields Value	Description
`Where`	Changes to fields referenced in the WHERE clause are evaluated. Notifications are generated for all records where a field referenced in the WHERE clause changes and the values of the fields referenced in the WHERE clause match the values specified in the WHERE clause.

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

`NotifyForFields` Set to `All`

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

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

Examples

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

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

Warning: Use caution when setting `NotifyForFields` to `All`. When you use this value, then notifications are generated for all record field changes as long as the new field values match the values in the WHERE clause. Therefore, the number of generated notifications could potentially be large, and you may hit the daily quota of events limit. 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 if a notification should be generated.

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

Event	A notification is generated when
Record is created	The record field values match the values specified in the WHERE clause
Record is updated	• A change occurs in one or more record fields that are specified in the PushTopic query SELECT clause or • A change occurs in one or more record fields that are specified in the PushTopic query WHERE clause and • The record values of the fields specified in the WHERE clause all match the values in the PushTopic query WHERE clause

Examples

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

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

`NotifyForFields` Set to `Select`

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

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

Event	A notification is generated when
Record is created	The record field values match the values specified in the WHERE clause
Record is updated	• A change occurs in one or more record fields that are specified in the PushTopic query SELECT clause and • The record values of the fields specified in the WHERE clause all match the values in the PushTopic query WHERE clause

Examples

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

PushTopic Query	Result
'e10R0000000KEU9IAO', 'v32B0000000KWZ7YEP')	

`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 if a notification should be generated.

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

Event	A notification is generated when
Record is created	The record field values match the values specified in the WHERE clause
Record is updated	• A change occurs in one or more record fields that are specified in the PushTopic query WHERE clause and • The record values of the fields specified in the WHERE clause all match the values in the PushTopic query WHERE clause

Examples

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

Notification Scenarios

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

Scenario	Configuration
You want to receive all notifications of all record updates.	• ***MyPushTopic***`.Query = SELECT Id, Name, Description__c FROM InvoiceStatement` • ***MyPushTopic***`.NotifyForFields = All`

Scenario	Configuration
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.	• `MyPushTopic.Query = SELECT Id, Name, Amount__c FROM InvoiceStatement` • `MyPushTopic.NotifyForFields = Referenced`
You want to receive notification of all record changes made to a specific record.	• `MyPushTopic.Query = SELECT Id, Name, Amount__c FROM InvoiceStatement WHERE Id='a07B0000000KWZ7IAO'` • `MyPushTopic.NotifyForFields = All`
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.	• `MyPushTopic.Query = SELECT Id, Name, Amount__c FROM InvoiceStatement WHERE Id='a07B0000000KWZ7IAO'` • `MyPushTopic.NotifyForFields = Referenced`
You want to receive notification for all invoice statement record changes for vendors in a particular state.	• `MyPushTopic.Query = SELECT Id, Name, Amount__c FROM InvoiceStatement WHERE BillingState__c = 'NY'` • `MyPushTopic.NotifyForFields = All`
You want to receive notification for all invoice statement record changes where the invoice amount is $1,000 or more.	• `MyPushTopic.Query = SELECT Id, Name FROM InvoiceStatement WHERE Amount > 999` • `MyPushTopic.NotifyForFields = Referenced`

Bulk Subscriptions

You can subscribe to multiple topics at the same time.

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

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

]
```

For more information, see the Bayeux Specification.

Deactivating a Push Topic

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

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

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

Chapter 6

Streaming API Considerations

In this chapter ...

Streaming API helps you create near real-time update notifications of your Database.com data. This chapter covers some client and troubleshooting considerations to keep in mind when implementing Streaming API.

Clients and Timeouts

Streaming API imposes three 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).

If there's no activity on that session, then the organization timeout goes into effect and closes the session.

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 Database.com 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 Browsers

Streaming API supports the following browsers:

- Internet Explorer 8 and greater
- Firefox 4 and greater

We recommend using the latest version of your browser with the most recent security updates and fixes applied. For regions that must use Internet Explorer 6 or 7, salesforce.com has confirmed that these browsers will work with Streaming API using jQuery 1.5.1 and CometD 2.2.0.

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 in order 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.

The first step for any debugging process is to follow the instructions in the Quick Start Using Workbench, or Example: Java Client and verify that you can implement the samples provided. The next step is to use your debug tool to help isolate the symptoms and cause of any problems.

402 Error

You may sometimes receive an error notification that contains “402::Uknown client” and looks something like this:

```
Thu Mar 29 06:08:08 PDT 2012 [CHANNEL:META_CONNECT]: {"id":"78","error":"402::Unknown
client","successful":false,"advice":{"interval":500,"reconnect":"handshake"}
```

This can be caused by various conditions including when your client connection times out. If you see this error, you should reconnect to the server with a handshake. For more information about client timeouts and Streaming API limits, see Clients and Timeouts and Streaming API Limits.

Monitoring Events Usage

The number of events that can be generated in a 24–hour period depends on your type of organization. For more information, see Streaming API Limits. You can monitor Streaming API events usage on the Company Information page.

- From Setup, click **Company Profile** > **Company Information**.

If you refresh the Company Information page, the Streaming API Events value may fluctuate slightly. Regardless of these small fluctuations, your limits are being assessed accurately.

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": "1g177wgjj14omtdo3rcl0hjhm4w",
  "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": "1g177wgjj14omtdo3rcl0hjhm4w",
  "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"
    }
  }
}
```

GENERIC STREAMING

Chapter 7

Introducing Generic Streaming

Generic streaming uses Streaming API to send notifications of general events that are not tied to Database.com data changes.

Use generic streaming when you want to send and receive notifications based on custom events that you specify. You can use generic streaming for any situation where you need to send custom notifications, such as:

- Broadcasting notifications to specific teams, or your entire organization.
- Sending notifications for events that are external to Database.com.

To use generic streaming, you need:

- A StreamingChannel that defines the channel.
- One or more clients subscribed to the channel.
- The Streaming Channel Push REST API resource that lets you monitor and invoke push events on the channel.

Note: Generic streaming is currently available through a pilot program. For information on enabling generic streaming for your organization, contact salesforce.com, inc.

Chapter 8

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.

Create a Streaming Channel

Create a new StreamingChannel object by using the Database.com UI.

Create a Java Client

Create a Java client that uses Bayeux and CometD to subscribe to the channel.

Generate Events Using REST

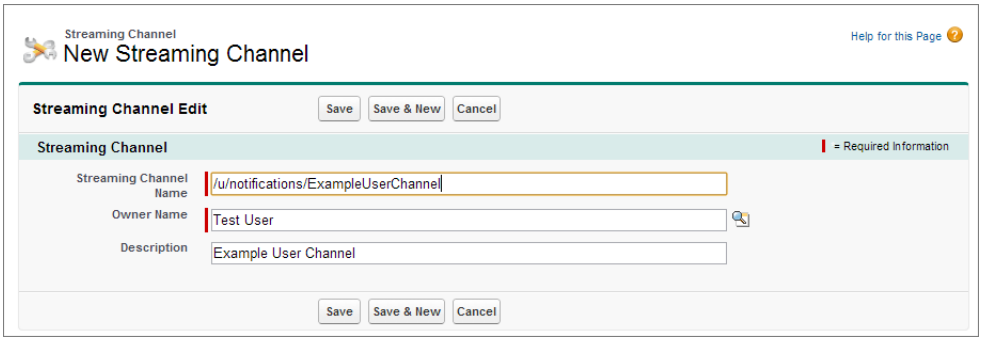
Use the Streaming Channel Push REST API resource to generate event notifications to channel subscribers.

Create a Streaming Channel

Create a new StreamingChannel object by using the Database.com UI.

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

1. Log into your Developer Edition organization. Under **All Tabs** (+) select **Streaming Channels**.
2. On the Streaming Channels tab, select **New** to create a new Streaming Channel.
3. Enter `/u/notifications/ExampleUserChannel` in **Streaming Channel Name**, and an optional description. Your new Streaming Channel page should look something like this:



4. Select **Save**. You've just created a new Streaming Channel that clients can subscribe to for notifications.

StreamingChannel is a regular, createable Database.com 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 will only send events 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, go to **Security Controls** > **Sharing Settings** and create or modify a StreamingChannel sharing rule.

Generic Streaming also supports dynamic streaming channel creation. With dynamic streaming channel creation, a StreamingChannel will be automatically created when a client first subscribes to the channel. To enable dynamic streaming channels in your organization, from Setup, go to **Customize** > **User Interface** and enable **Enable Dynamic Streaming Channel Creation**.

Create a Java Client

Create a Java client that uses Bayeux and CometD to subscribe to the channel.

1. Download and install the CometD and Jetty .jar files if necessary.
2. In a new Java project, add the following code to a Java source file named `StreamingClientExample.java`. This code subscribes to the Streaming channel you created and listens for notifications. Depending on your Java development environment, you might have to rename this file and class to Main.

```
package demo;

import org.cometd.bayeux.Channel;
import org.cometd.bayeux.Message;
import org.cometd.bayeux.client.ClientSessionChannel;
import org.cometd.bayeux.client.ClientSessionChannel.MessageListener;
import org.cometd.client.BayeuxClient;
import org.cometd.client.transport.ClientTransport;
import org.cometd.client.transport.LongPollingTransport;

import org.eclipse.jetty.client.ContentExchange;
import org.eclipse.jetty.client.HttpClient;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;

/**
 * This example demonstrates how a streaming client works
 * against the Salesforce Streaming API with generic notifications.
 **/

public class StreamingClientExample {

    // This URL is used only for logging in. The LoginResult
    // returns a serverUrl which is then used for constructing
    // the streaming URL. The serverUrl points to the endpoint
    // where your organization is hosted.

    static final String LOGIN_ENDPOINT = "https://login.database.com";
   private static final String USER_NAME = "change_this_to_your_testuser@yourcompany.com";

    private static final String PASSWORD = "change_this_to_your_testpassword";
    // NOTE: Putting passwords in code is not a good practice and not recommended.

    // The channel to subscribe to.
    // Be sure to create the StreamingChannel before running this sample.
    private static final String CHANNEL = "/u/notifications/ExampleUserChannel";
    private static final String STREAMING_ENDPOINT_URI = "/cometd/30.0";

    // The long poll duration.
    private static final int CONNECTION_TIMEOUT = 20 * 1000;  // milliseconds
    private static final int READ_TIMEOUT = 120 * 1000; // milliseconds
```

```
    public static void main(String[] args) throws Exception {

     System.out.println("Running streaming client example....");

        final BayeuxClient client = makeClient();
        client.getChannel(Channel.META_HANDSHAKE).addListener
            (new ClientSessionChannel.MessageListener() {

            public void onMessage(ClientSessionChannel channel, Message message) {

             System.out.println("[CHANNEL:META_HANDSHAKE]: " + message);

                boolean success = message.isSuccessful();
                if (!success) {
                    String error = (String) message.get("error");
                    if (error != null) {
                        System.out.println("Error during HANDSHAKE: " + error);
                        System.out.println("Exiting...");
                        System.exit(1);
                    }

                    Exception exception = (Exception) message.get("exception");
                    if (exception != null) {
                        System.out.println("Exception during HANDSHAKE: ");
                        exception.printStackTrace();
                        System.out.println("Exiting...");
                        System.exit(1);

                    }
                }
            }

        });

        client.getChannel(Channel.META_CONNECT).addListener(
            new ClientSessionChannel.MessageListener() {
            public void onMessage(ClientSessionChannel channel, Message message) {

                System.out.println("[CHANNEL:META_CONNECT]: " + message);

                boolean success = message.isSuccessful();
                if (!success) {
                    String error = (String) message.get("error");
                    if (error != null) {
                        System.out.println("Error during CONNECT: " + error);
                        System.out.println("Exiting...");
                        System.exit(1);
                    }
                }
            }

        });

        client.getChannel(Channel.META_SUBSCRIBE).addListener(
            new ClientSessionChannel.MessageListener() {

            public void onMessage(ClientSessionChannel channel, Message message) {

             System.out.println("[CHANNEL:META_SUBSCRIBE]: " + message);
                boolean success = message.isSuccessful();
                if (!success) {
                    String error = (String) message.get("error");
                    if (error != null) {
                        System.out.println("Error during SUBSCRIBE: " + error);
                        System.out.println("Exiting...");
                        System.exit(1);
                    }
                }
            }
        });
```

```
        client.handshake();
        System.out.println("Waiting for handshake");

        boolean handshaken = client.waitFor(10 * 1000, BayeuxClient.State.CONNECTED);
        if (!handshaken) {
            System.out.println("Failed to handshake: " + client);
            System.exit(1);
        }



        System.out.println("Subscribing for channel: " + CHANNEL);

        client.getChannel(CHANNEL).subscribe(new MessageListener() {
            @Override
            public void onMessage(ClientSessionChannel channel, Message message) {
                System.out.println("Received Message: " + message);
            }
        });

        System.out.println("Waiting for streamed data from your organization ...");
        while (true) {
            // This infinite loop is for demo only,
            // to receive streamed events on the
            // specified topic from your organization.
        }
    }



    private static BayeuxClient makeClient() throws Exception {
        HttpClient httpClient = new HttpClient();
        httpClient.setConnectTimeout(CONNECTION_TIMEOUT);
        httpClient.setTimeout(READ_TIMEOUT);
        httpClient.start();

        String[] pair = SoapLoginUtil.login(httpClient, USER_NAME, PASSWORD);

        if (pair == null) {
            System.exit(1);
        }

        assert pair.length == 2;
        final String sessionid = pair[0];
        String endpoint = pair[1];
        System.out.println("Login successful!\nEndpoint: " + endpoint
            + "\nSessionid=" + sessionid);

        Map<String, Object> options = new HashMap<String, Object>();
        options.put(ClientTransport.TIMEOUT_OPTION, READ_TIMEOUT);
        LongPollingTransport transport = new LongPollingTransport(
          options, httpClient) {

         @Override
            protected void customize(ContentExchange exchange) {
                super.customize(exchange);
                exchange.addRequestHeader("Authorization", "OAuth " + sessionid);
            }
        };

        BayeuxClient client = new BayeuxClient(salesforceStreamingEndpoint(
            endpoint), transport);
        return client;
    }

    private static String salesforceStreamingEndpoint(String endpoint)
        throws MalformedURLException {
```

```
        return new URL(endpoint + STREAMING_ENDPOINT_URI).toExternalForm();
    }

}
```

3. Edit `StreamingClientExample.java` and modify the following values:

File Name	Static Resource Name
USER_NAME	Username of the logged-in user
PASSWORD	Password for the *USER_NAME* (or logged-in user)
CHANNEL	`/u/notifications/ExampleUserChannel`
LOGIN_ENDPOINT	`https://test.database.com` (Only if you are using a test database. If you are in a production organization, no change is required for LOGIN_ENDPOINT.)

4. Add the following code to a Java source file named `SoapLoginUtil.java`. This code sends a username and password to the server and receives the session ID.

Important: Never handle the usernames and passwords of others. Before using in a production environment, delegate the login to OAuth.

```
package demo;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.MalformedURLException;
import java.net.URL;

import org.eclipse.jetty.client.ContentExchange;
import org.eclipse.jetty.client.HttpClient;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

public final class SoapLoginUtil {

    // The enterprise SOAP API endpoint used for the login call in this example.
   private static final String SERVICES_SOAP_PARTNER_ENDPOINT = "/services/Soap/u/22.0/";


    private static final String ENV_START =
            "<soapenv:Envelope xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope/'
 "
                    + "xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' " +
                    "xmlns:urn='urn:partner.soap.sforce.com'><soapenv:Body>";

    private static final String ENV_END = "</soapenv:Body></soapenv:Envelope>";

    private static byte[] soapXmlForLogin(String username, String password)
        throws UnsupportedEncodingException {
        return (ENV_START +
                "  <urn:login>" +
                "    <urn:username>" + username + "</urn:username>" +
                "    <urn:password>" + password + "</urn:password>" +
                "  </urn:login>" +
```

```
                ENV_END).getBytes("UTF-8");
    }

    public static String[] login(HttpClient client, String username, String password)
            throws IOException, InterruptedException, SAXException,
                ParserConfigurationException {

        ContentExchange exchange = new ContentExchange();
        exchange.setMethod("POST");
        exchange.setURL(getSoapURL());
        exchange.setRequestContentSource(new ByteArrayInputStream(soapXmlForLogin(
            username, password)));
        exchange.setRequestHeader("Content-Type", "text/xml");
        exchange.setRequestHeader("SOAPAction", "''");
        exchange.setRequestHeader("PrettyPrint", "Yes");

        client.send(exchange);
        exchange.waitForDone();
        String response = exchange.getResponseContent();

        SAXParserFactory spf = SAXParserFactory.newInstance();
        spf.setNamespaceAware(true);
        SAXParser saxParser = spf.newSAXParser();

        LoginResponseParser parser = new LoginResponseParser();
        saxParser.parse(new ByteArrayInputStream(
            response.getBytes("UTF-8")), parser);

        if (parser.sessionId == null || parser.serverUrl == null) {
            System.out.println("Login Failed!\n" + response);
            return null;
        }

        URL soapEndpoint = new URL(parser.serverUrl);
        StringBuilder endpoint = new StringBuilder()
            .append(soapEndpoint.getProtocol())
            .append("://")
            .append(soapEndpoint.getHost());

        if (soapEndpoint.getPort() > 0) endpoint.append(":")
            .append(soapEndpoint.getPort());
        return new String[] {parser.sessionId, endpoint.toString()};
    }

    private static String getSoapURL() throws MalformedURLException {
        return new URL(StreamingClientExample.LOGIN_ENDPOINT +
            getSoapUri()).toExternalForm();
    }

    private static String getSoapUri() {
        return SERVICES_SOAP_PARTNER_ENDPOINT;
    }

    private static class LoginResponseParser extends DefaultHandler {

        private boolean inSessionId;
        private String sessionId;

        private boolean inServerUrl;
        private String serverUrl;

        @Override
        public void characters(char[] ch, int start, int length) {
            if (inSessionId) sessionId = new String(ch, start, length);
            if (inServerUrl) serverUrl = new String(ch, start, length);
        }

        @Override
        public void endElement(String uri, String localName, String qName) {
            if (localName != null) {
```

```
                if (localName.equals("sessionId")) {
                    inSessionId = false;
                }

                if (localName.equals("serverUrl")) {
                    inServerUrl = false;
                }
            }
        }

        @Override
        public void startElement(String uri, String localName,
            String qName, Attributes attributes) {
            if (localName != null) {
                if (localName.equals("sessionId")) {
                    inSessionId = true;
                }

                if (localName.equals("serverUrl")) {
                    inServerUrl = true;
                }
            }
        }
    }
}
```

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

```
Running streaming client example....
Login successful!
Endpoint: https:www.database.com
Sessionid=00DD0000000FSp9!AQIAQIVjGYijFhiAROTc455T6kEVeJGXuW5VCnp
    LANCMawS7.p5fXbjYlqCgx7They_zFjmP5n9HxvfUA6xGSGtC1Nb6P4S.

Waiting for handshake
[CHANNEL:META_HANDSHAKE]:
{
    "id":"1",
    "minimumVersion":"1.0",
    "supportedConnectionTypes":["long-polling"],
    "successful":true,
    "channel":"/meta/handshake",
    "clientId":"31t0cjzfbgnfqn1rggumba0k98u",
    "version":"1.0"
}

[CHANNEL:META_CONNECT]:
{
    "id":"2",
    "successful":true,
    "advice":{"interval":0,"reconnect":"retry","timeout":110000},
    "channel":"/meta/connect"}
    Subscribing for channel: /u/notifications/ExampleUserChannel
    Waiting for streamed data from your organization ...
[CHANNEL:META_SUBSCRIBE]:
{
    "id":"4",
    "subscription":"/u/notifications/ExampleUserChannel",
    "successful":true,
    "channel":"/meta/subscribe"
}

[CHANNEL:META_CONNECT]:
{
    "id":"3",
    "successful":true,
    "channel":"/meta/connect"
}
```

```
Received Message:
{
    "data":
    {
      "event":
      {
        "createdDate":"2013-07-30T23:15:59.000+0000"
      },
      "payload":"Broadcast message to all subscribers"
    },
    "channel":"/u/notifications/ExampleUserChannel",
    "clientId":"8173z2cplh8q6m1rmud93zygnf8"
}

[CHANNEL:META_CONNECT]:
{
    "id":"5",
    "successful":true,
    "channel":"/meta/connect"
}
```

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.com provides a hosted instance of Workbench for demonstration purposes only—salesforce.com 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.

1. In a browser, navigate to `http://workbench.developerforce.com`.
2. For Environment, select **Production**.
3. For API Version, select 30.0.
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/v29.0/sobjects/StreamingChannel/`**`Streaming Channel ID`**`/push`, where `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 POST REST request body:

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

```
            "userIds": []
        }
    ]
}
```

Chapter 9

PushTopic

Represents a query that is the basis for notifying listeners of changes to records in an organization. This is available from API version 21.0 or later.

Supported Calls

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

SOAP: `create()`, `delete()`, `describe()`, `describeSObjects()`, `query()`, `retrieve()`, `update()`

Special Access Rules

- This object is only available if Streaming API is enabled for your organization.
- Only users with "Create" permission can create this record. Users with "View All Data" can view PushTopic records and see streaming messages.

Fields

Field	Field Type	Description
`ApiVersion`	double	Required. API version to use for executing the query specified in `Query`. It must be an API version greater than 20.0. Example value: 30.0 **Field Properties:** Create, Filter, Sort, Update
`Description`	string	Description of the PushTopic. Limit: 400 characters **Field Properties:** Create, Filter, Sort, Update
`ID`	ID	System field: Globally unique string that identifies a record. **Field Properties:** Default on create, Filter, Group, idLookup, Sort
`isActive`	boolean	Indicates whether the record currently counts towards the organization's limit. **Field Properties:** Create, Default on create, Filter, Group, Sort, Update
`IsDeleted`	boolean	System field: Indicates whether the record has been moved to the Recycle Bin (`true`) or not (`false`). **Field Properties:** Default on create, Filter, Group, Sort

Field	Field Type	Description
`Name`	string	Required. Descriptive name of the PushTopic. Limit: 25 characters. This value identifies the channel. **Field Properties:** Create, Filter, Group, Sort, Update
`NotifyForFields`	picklist	Specifies which fields are evaluated to generate a notification. Valid values: • `All` • `Referenced` (default) • `Select` • `Where` **Field Properties:** Create, Filter, Sort, Update
`NotifyForOperations`	picklist	Specifies which record events may generate a notification. Valid values: • `All` (default) • `Create` • `Extended` • `Update` **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.
`NotifyForOperationCreate`	boolean	`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.
`NotifyForOperationDelete`	boolean	`true` if a delete operation should generate a notification, otherwise,`false`. Defaults to `true`. Clients must connect using the `cometd/29.0` (or later) Streaming API endpoint to receive delete and undelete event notifications. This field is available in API version 29.0 and later.
`NotifyForOperationUndelete`	boolean	`true` if an undelete operation should generate a notification, otherwise, `false`. Defaults to `true`. Clients must connect using the `cometd/29.0` (or later) Streaming API endpoint to receive delete and undelete event notifications. This field is available in API version 29.0 and later.
`NotifyForOperationUpdate`	boolean	`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.
`Query`	string	Required. The SOQL query statement that determines which record changes trigger events to be sent to the channel. Limit: 1300 characters **Field Properties:** Create, Filter, Sort, Update

`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

Chapter 10

StreamingChannel

Represents a channel that is the basis for notifying listeners of generic Streaming API events. This is available from API version 29.0 or later.

Note: Generic streaming is currently available through a pilot program. For information on enabling generic streaming for your organization, contact salesforce.com, inc..

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 only available if Streaming API is enabled for your organization.
- Only users with "Create" permission can create this record. Users with "View All Data" can view StreamingChannel records and see streaming messages.
- 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 will only send events 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.

Fields

Field	Field Type	Description
`Description`	string	Description of the StreamingChannel. Limit: 255 characters. **Field Properties:** Create, Filter, Group, Nillable, Sort, Update **Label:** Description
`ID`	ID	System field: Globally unique string that identifies a StreamingChannel record. **Field Properties:** 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`). **Field Properties:** Default on create, Filter, Group, Sort
`IsDynamic`	boolean	`true` if the channel gets dynamically created on subscribe if necessary, `false` otherwise. To enable dynamic streaming channels in your organization, from Setup,

Field	Field Type	Description
		go to **Customize** > **User Interface** and enable **Enable Dynamic Streaming Channel Creation**. **Field Properties:** Default on create, Filter, Group, Sort
`LastReferencedDate`	date	The timestamp for when the current user last viewed a record related to this record. **Field Properties:** Filter, Sort
`LastViewedDate`	date	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. **Field Properties:** Filter, Sort
`Name`	string	Required. Descriptive name of the StreamingChannel. Limit: 80 characters, alphanumeric and "_", "/" characters only. Must start with "/u/". This value identifies the channel. **Field Properties:** Create, Filter, Group, idLookup, Sort, Update **Label:** Streaming Channel Name
`OwnerId`	reference	The ID of the owner of the StreamingChannel. **Field Properties:** Create, Default on create, Filter, Group, Sort, Update **Label:** Owner Name

Chapter 11

Streaming Channel Push

Gets subscriber information and pushes notifications for Streaming Channels.

Note: Generic streaming is currently available through a pilot program. For information on enabling generic streaming for your organization, contact salesforce.com, inc.

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:

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

Each push event payload contains the following fields:

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

Request parameters

None

Response data

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

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

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

Name	Type	Description
`fanoutCount`	number	The number of subscribers that the event got sent to. This is the count of subscribers specified in the POST request that were online. If the request was broadcast to all subscribers, fanoutCount will be –1. If no active subscribers were found for the channel, fanoutCount will be 0.
`userOnlineStatus`	array of User online status information	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`.

Example

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

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

Using a POST request to `services/data/v29.0/sobjects/StreamingChannel/0M6D000000000g7KXA/push` with a request JSON body of:

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

the JSON response data looks something like:

```
[
  {
    "fanoutCount" : 1,
    "userOnlineStatus" : {
```

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

Chapter 12

Streaming API Limits

Limits protect shared resources. These are the default limits intended for basic consumers of Streaming API. If your application exceeds these limits, or you have scenarios where you need to increase the number of clients per topic or the number of concurrent clients across all topics, please contact salesforce.com.

Description	Limit
Maximum number of topics (PushTopic records) per organization	40
Maximum number of clients (subscribers) per topic	20
Maximum number of concurrent clients (subscribers) across all topics	20
Maximum number of events per day (24–hour period)	50,000 for all organizations except free organizations, where the maximum is 10,000
Socket timeout during connection (CometD session)	110 seconds
Timeout to reconnect after successful connection (keepalive)	40 seconds
Maximum length of the SOQL query in the `Query` field of a PushTopic record	1300 characters
Maximum length for a PushTopic name	25 characters

Generic Streaming Limits

The following limits apply to generic streaming.

Note: Generic streaming is currently available through a pilot program. For information on enabling generic streaming for your organization, contact salesforce.com, inc.

Description	Limit
Maximum number of StreamingChannels per organization	1000
Maximum number of events per day (24–hour period)	100,000

The limits on maximum number of clients and maximum number of concurrent clients for generic streaming are the same limits used for PushTopic streaming.

Description	Limit
Maximum number of clients (subscribers) per generic streaming channel	20
Maximum number of concurrent clients (subscribers) across all generic streaming channels	20

Index

S

T

U