International Journal of Science and Research (IJSR)
ISSN: 2319-7064
ResearchGate Impact Factor (2018): 0.28 | SJIF (2018): 7.426

How to Subscribe to Google Pub/Sub Topic from Salesforce using Push Method with Spring Boot Java Application

Chirag Amrutlal Pethad

PetSmart.com, LLC, Stores and Services
Phoenix, Arizona, USA
Email: *ChiragPethad[at]live.com, ChiragPethad[at]gmail.com, Cpethad[at]petsmart.com*

Abstract: ***The document outlines the integration of Google Cloud Pub/Sub with Salesforce using a push mechanism by creating a spring boot Java application. Key steps include setting up a Pub-Sub topic and subscription, creating an Apex REST service in Salesforce to handle incoming messages, also creating a spring boot Java application to subscribe to the topic and forward the messages to APEX REST service, and implementing OAuth 2.0 for secure authentication. It emphasizes security considerations, testing, and best practices for error handling and scalability, ultimately enhancing Salesforce applications' responsiveness and reliability through real-time messaging.***

Keywords: Event Bus, Event Driven Architecture, Google PUB-SUB, Integration, Push vs Pull, REST API, Limits, Scalability, Event Replay, Spring boot

1. Introduction

Salesforce Event Bus, also known as the Platform Events framework, is a powerful tool for enabling event-driven architectures within Salesforce. However, there are several limitations to consider when using Salesforce Event Bus. The integration of Google Cloud Pub/Sub with Salesforce enables businesses to harness the power of real-time data processing and avoid the limitations associated with Salesforce Event Bus. This white paper provides a step-by-step guide to setting up and subscribing to Google Pub/Sub in Salesforce using Push method leveraging the capabilities of both platforms for improved operational efficiency, seamless and efficient flow of information. The objective is to enable real-time messaging and event-driven architecture in Salesforce by leveraging GCP Pub/Sub for asynchronous communication. This integration allows Salesforce to automatically receive messages pushed from Pub/Sub topics, ensuring efficient and scalable processing of events.

2. Limitations of Salesforce Event Bus

Salesforce Event Bus, also known as the Platform Events framework, is a powerful tool for enabling event-driven architectures within Salesforce and integrating with external systems. However, there are several limitations to consider when using Salesforce Event Bus:

1) Event Delivery

a) No Guaranteed Order: While Salesforce attempts to deliver events in order, it does not guarantee the order of event delivery.
b) At-Least-Once Delivery: Events may be delivered more than once. Consumers must handle potential duplicate events.

2) Event Retention and Replay

a) Retention Period: Platform events are retained for 72 hours. If consumers are offline for longer than this period, they may miss events.
b) Limited Replay Options: Replay of events is limited to the last 24 hours. For events older than 24 hours but within the 72-hour retention period, consumers must handle gaps manually.

3) Event Size and Volume

a) Payload Size: The maximum size of a platform event message is 1 MB. This includes the payload and metadata.
b) Volume Limits: There are limits on the number of events that can be published and delivered within a 24-hour period, depending on the Salesforce edition and licensing:
- There are limits on the number of events that can be published and delivered within a 24-hour period, depending on the Salesforce edition and licensing.
- Standard Volume Platform Events: 50,000 events per 24-hour period.

4) Event Publishing Limits

There are limits on the number of events that can be published per transaction and per hour. Exceeding these limits will result in errors:

- Per Transaction: 1,000 events
- Per Hour: Limits vary by Salesforce edition and license count.

5) Event Processing Limits

- Subscriber Limits: Each event can have a maximum of 50 subscribers (including Apex triggers, flows, and external systems).
- Concurrency Limits: Salesforce imposes limits on the number of concurrent long-running Apex transactions, which can impact event processing performance.

6) Platform Events and Triggers

- Governor Limits: Apex triggers on platform events are subject to Salesforce governor limits, such as CPU time, heap size, and SOQL/DML limits.
- Error Handling: Errors in triggers can cause event processing failures. Proper error handling and retry mechanisms must be implemented.

7) Integration and External Systems

- External System Dependencies: Integrating with external systems can introduce latency and reliability issues. Ensure that external systems can handle the volume and frequency of events.
- API Limits: Calling external APIs from Salesforce is subject to API call limits and rate limits imposed by the external system.

8) Maintenance and Upgrades

- API Versioning: Changes to Salesforce API versions can impact event processing. Ensure compatibility with the latest API versions.
- Platform Upgrades: Salesforce platform upgrades may introduce changes that impact event bus functionality. Monitor release notes and perform testing during upgrades.

9) Monitoring and Debugging

- Limited Monitoring Tools: Salesforce provides limited built-in tools for monitoring platform events. Additional third-party tools or custom monitoring solutions may be needed for comprehensive monitoring and alerting.
- Debugging Challenges: Debugging issues with event processing can be challenging due to asynchronous nature and potential delays in event delivery.

3. Understanding Google Cloud Pub Sub

Google Cloud Pub/Sub is a fully-managed real-time messaging service that allows you to send and receive messages between independent applications. It decouples services that produce events from services that process events, enhancing scalability and reliability.

a) Key Features

- Scalability: Handle high throughput and low-latency messages.
- Reliability: Ensure message delivery with at-least-once delivery.
- Flexibility: Integrate with various GCP services and external systems.

b) Overview of Salesforce Apex

Apex is a strongly-typed, object-oriented programming language used by developers to execute flow and transaction control statements on the Salesforce platform. It enables the creation of web services, email services, and complex business processes.

c) Key Features

- Scalability: Handle large volumes of data and transactions.
- Robustness: Build complex logic and automation workflows.
- Integration Capabilities: Interact with external systems via REST and SOAP APIs.

d) Overview of Salesforce Apex

Spring Boot is an extension of the Spring framework that simplifies the setup and development of new Spring applications. It provides a range of features and tools for building production-ready applications with minimal configuration. Spring Boot aims to simplify the development of Spring-based applications by providing a convention-over-configuration approach. It offers a range of pre-configured components and sensible defaults, reducing the need for extensive setup and boilerplate code. Some of the Key Features of Spring Boot includes:

- Auto-Configuration: Automatically configures Spring and third-party libraries whenever possible.
- Standalone: Creates standalone applications with embedded servers (e.g., Tomcat, Jetty) that can be run with “java -jar” command.
- Production-Ready: Includes production-ready features such as metrics, health checks, and externalized configuration.
- Opinionated Defaults: Provides opinionated defaults for application setup to speed up development.
- Micro-services Support: Ideal for building micro-services with features like embedded servers, health checks, and distributed tracing.

4. Implementation Plan

The implementation plan involves setting up a Google Cloud Pub/Sub topic, configuring service accounts and permissions, and implementing Java application to subscribe to the Pub/Sub topic and then publish / forward those messages to Salesforce endpoint. The process includes:

a) Setting up Google Cloud Pub/Sub
Create a Pub-Sub topic and configure necessary IAM roles.

b) Salesforce Setup
Create a web service to receive messages.

c) Create a Spring Boot Java Application
Develop Java application that Subscribes to Pub/Sub and push messages to the Salesforce endpoint.

5. Step by Step Implementation

a) Create a Pub/Sub Topic

- Go to Google Cloud Console
- Navigate to Pub/Sub section
- Create a New Topic

b) Create a Subscription

- In the Google Cloud Console, navigate to the Pub-Sub topic you created.
- Create a new subscription and select the "Pull" delivery type.

c) Configure IAM permissions

- Step 1: Create a Service Account
- Navigate to IAM & Admin > Service Accounts.
- Click + CREATE SERVICE ACCOUNT.
- Enter a name and description for the service account, then click CREATE.
- Assign the required roles to the service account, such as Pub/Sub Subscriber.
- Step 2: Create a Key for the Service Account
- In the Service Accounts page, find your new service account.
- Click the Actions column (three dots) for your service account and select Manage keys.
- Click ADD KEY > Create new key.

- Select JSON and click Create to download the JSON key file.

d) Create a Service Account User with required permissions in Salesforce

- Step 1: Create New User
- Log in to your Salesforce instance with an account that has administrative privileges.
- Click on the gear icon in the top right corner to open the Setup menu.
- In the Quick Find box on the left, type Users and select Users under the Users section.
- Click on the New User button.
  - First Name: Service (or a name indicating the account's purpose)
  - Last Name: Account
  - Email: Provide a valid email address (for notifications and password reset)
  - Username: Must be in the format of an email address (unique across all Salesforce instances)
  - User License: Select the appropriate license type, typically Salesforce or API Only.
  - Profile: Assign a profile that provides the necessary permissions (e.g., System Administrator or a custom profile with specific API permissions).

- Ensure the Generate new password and notify user immediately option is checked if you want the login credentials to be sent to the email address specified. Click Save.

  Step 2: Assign Permissions
  - After creating the user, you may need to assign additional permissions via Permission Sets to grant specific access required for the service account.
  - In the Setup menu, type Permission Sets in the Quick Find box and select Permission Sets.
  - Create a new Permission Set or use an existing one.
  - Assign the necessary permissions (e.g., API access, specific object permissions).
  - Go to the Users section of the Permission Set and assign the newly created service account user to this Permission Set.
  - If the service account will be used for API access, ensure the profile or permission set assigned to the user includes API Enabled permissions.

- Navigate to the System Permissions section within the Profile or Permission Set and ensure API Enabled is checked.

  Step 3: Set-Up Connected App
  - If the service account will be used with OAuth for authentication, you need to create a Connected App.
  - In the Setup menu, type App Manager in the Quick Find box and select App Manager.
  - Click New Connected App.
  - Fill in the required fields, such as Connected App Name, API Name, Contact Email.
  - Under API (Enable OAuth Settings), check the Enable OAuth Settings box.
  - Provide the Callback URL and select the necessary OAuth scopes (e.g., Full Access, Perform requests on your behalf at any time).
  - Save the Connected App and note the Consumer Key and Consumer Secret for API integration.

e) Create and Configure Salesforce Service to Receive Pub/Sub Messages

Step 1: Create a Salesforce APEX REST Service
- In Salesforce, navigate to Setup.
- Go to Apex Classes and click "New" to create a new Apex class that will handle incoming Pub/Sub messages.

```
//@RestResource(urlMapping='/PubSubHandler/*')
global with sharing class PubSubHandler {
    @HttpPost
    global static void doPost() {
        RestRequest req = RestContext.request;
        RestResponse res = RestContext.response;

        String requestBody = req.requestBody.toString();
        Map<String, Object> message = (Map<String, Object>)
              fiJSON.deserializeUntyped(requestBody);
        String messageData = (String) message.get('message').get('data');

        Blob decodedBlob = EncodingUtil.base64Decode(messageData);
        String decodedMessage = decodedBlob.toString();

        processMessage(decodedMessage);

        res.responseBody = Blob.valueOf('Message processed successfully');
        res.statusCode = 200;
    }

    private static void processMessage(String message) {
        // Implement message processing logic
    }
}
```

- Grant access to PubSubHandler for the Service Account

f) Create and Configure Spring Boot Java Application

Step 1: Configure and Create the Project
- Open Spring Initializr. - https://start.spring.io/
- Project: Select "Maven Project"
- Language: Select "Java" and the version you want to use.
- Spring Boot: Choose the version of Spring Boot you want to use.
- Project Metadata:
  - Group: com.example
  - Artifact: demo, Name: demo
  - Description: Demo project for Spring Boot
  - Package name: com.example.demo
  - Packaging: Jar
- Add Dependencies
  - Spring Web
  - Google Cloud Messaging
  - Spring Boot DevTools
  - Spring Integration
- Generate and Download the Project: Click on the "Generate" button. A ZIP file will be downloaded. Extract this ZIP file to your desired location.
- Import the Project into your IDE: Open your preferred IDE (e.g., IntelliJ IDEA, Eclipse, Visual Studio Code), and import the project as a Maven project.
- Add Additional dependencies to “pom.xml” file in the project:

```
<properties>
  <spring-cloud-gcp.version>1.2.5.RELEASE</spring-cloud-gcp.version>
</properties>

<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
</dependency>

<dependency>
    <groupId>com.force.api</groupId>
    <artifactId>force-wsc</artifactId>
    <version>38.0.0</version>
</dependency>
<dependency>
    <groupId>com.force.api</groupId>
    <artifactId>force-partner-api</artifactId>
    <version>38.0.0</version>
</dependency>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-gcp-dependencies</artifactId>
      <version>${spring-cloud-gcp.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- Configure Application / Environment Properties

```
# Google Cloud Project Id
spring.cloud.gcp.project-id=mygcpproject-id12344
# Google Cloud Credentials Json file classpath location
spring.cloud.gcp.credentials.location=classpath:=mygcpproject-id12344-aas13123123.json

# Google Pub Sub Subscription name
apex.subscription_name=PubSubSubcriptionName

#Salesforce credentials for Service Account and Connected App
apex.token_url=https://login.salesforce.com/services/oauth2/token
apex.username=SalesforceServiceAccountUsername
apex.password=SaleforcePasswordandToken
apex.client_secret=ConnectedAppSecret
apex.client_id=ConnectedAppClientID
```

Step 2: Develop the Spring Boot Application

- First, we create the Main Class of our application. This class should get autogenerated when we download the project zip file.

```
package <<package-name>>;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyPubSubApplication {

   public static void main(String[] args) {
    SpringApplication.run(MyPubSubApplication.class, args);
  }
}
```

- Constant Interface for all environment variables and Constants.

```
package <<package-name>>;

public interface AppConstants {
  public static final String USERNAME = "username";
  public static final String PASSWORD = "password";
  public static final String CLIENT_SECRET = "client_secret";
  public static final String CLIENT_ID = "client_id";
  public static final String GRANT_TYPE = "grant_type";
  public static final String CONTENT_TYPE = "contentType";
  public static final String ENV_USERNAME = "apex.username";
  public static final String ENV_PASSWORD = "apex.password";
  public static final String ENV_CLIENT_SECRET = "apex.client_secret";
  public static final String ENV_CLIENT_ID = "apex.client_id";
  public static final String GRANT_TYPE_PASSWORD = "password";
  public static final String CONTENT_TYPE_FORM = "application/x-www-form-urlencoded";
  public static final String ENV_TOKEN_URL = "apex.token_url";
  public static final String ENV_SUBSCRIPTION_NAME = "apex.subscription_name";
}
```

- Next, we create a Service for handling the Salesforce Login to get the access token.

```
package <<package-name>>;

public class AuthenticationResponse {
    private String access_token;
    private String instance_url;
    private String token_type;
    private String issued_at;

    //TODO : Define getters and setters for all the attributes
}
```

```
package <<package-name>>;

import static <<package-name>>.AppConstants.*;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.util.StringUtils;
import org.springframework.web.client.RestTemplate;

@Component
public class ApexLoginService {

  @Autowired
  private Environment env;

  @Autowired
  private RestTemplate restTemplate;

  public AuthenticationResponse login(){
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
    MultiValueMap<String, String> params= new LinkedMultiValueMap<String, String>();
    params.add(USERNAME, getEnvProp(ENV_USERNAME));
    params.add(PASSWORD, getEnvProp(ENV_PASSWORD));
    params.add(CLIENT_SECRET, getEnvProp(ENV_CLIENT_SECRET));
    params.add(CLIENT_ID, getEnvProp(ENV_CLIENT_ID));
    params.add(GRANT_TYPE,GRANT_TYPE_PASSWORD);
    params.add(CONTENT_TYPE,CONTENT_TYPE_FORM);
    HttpEntity<MultiValueMap<String, String>> request
        = new HttpEntity<MultiValueMap<String, String>>(params, headers);
    ResponseEntity<AuthenticationResponse> response = restTemplate.postForEntity(
        getEnvProp(ENV_TOKEN_URL), request, AuthenticationResponse.class);
    return (AuthenticationResponse)response.getBody();
  }

  private String getEnvProp(String propertyName) {
    if(StringUtils.hasText(this.env.getRequiredProperty(propertyName))) {
      return this.env.getRequiredProperty(propertyName);
    }
    return "";
  }
}
```

- Next, we write a Service to Call the Salesforce REST endpoint to publish the message

```
package <<package-name>>;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.web.client.RestTemplate;

@Component
public class ApexEventService {

  @Autowired
  private RestTemplate restTemplate;

  public void publishApexEvent(AuthenticationResponse authResponse,
          String eventMessage) {
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);
    headers.setBearerAuth(authResponse.getAccess_token());
    HttpEntity<ApexRequest> request = new HttpEntity<>(
        new ApexRequest(eventMessage), headers);
    String postEventUrl = authResponse.getInstance_url()
      +"/services/apexrest/PubSubHandler/";
    ResponseEntity<String> response = restTemplate.postForEntity(
      postEventUrl, request, String.class);
  }

  private class ApexRequest {
    private String message;

    public ApexRequest(String msg){
      this.message = msg;
    }
  }

}
```

- Finally, we create a service class that subscribes to the Pub-Sub Topic and then calls the ApexEventService to publish / push the message to Salesforce for further processing.

```
package <<package-name>>;

import static <<package-name>>.AppConstants.*;

@Configuration
public class PubSubHandlerService {

  @Autowired
  private Environment env;
  @Autowired
  private ApexLoginService apexLoginService;
  @Autowired
  private ApexEventService apexEventService;
  @Bean
  public MessageChannel pubsubInputChannel() {
    return new DirectChannel();
  }

  @Bean
  public PubSubInboundChannelAdapter messageChannelAdapter(
      @Qualifier("pubsubInputChannel") MessageChannel inputChannel,
        PubSubTemplate pubSubTemplate) {
    String subscriptionName = "";
    if(StringUtils.hasText(this.env.getRequiredProperty(ENV_SUBSCRIPTION_NAME))) {
      subscriptionName = this.env.getRequiredProperty(ENV_SUBSCRIPTION_NAME);
    }
    PubSubInboundChannelAdapter adapter = new PubSubInboundChannelAdapter(pubSubTemplate,
        subscriptionName);
    adapter.setOutputChannel(inputChannel);
    adapter.setAckMode(AckMode.MANUAL);
    return adapter;
  }

  @Bean
  @ServiceActivator(inputChannel = "pubsubInputChannel")
  public MessageHandler messageReceiver() {
    return message -> {
      BasicAcknowledgeablePubsubMessage originalMessage = message.getHeaders()
          .get(GcpPubSubHeaders.ORIGINAL_MESSAGE, BasicAcknowledgeablePubsubMessage.class);
      boolean apexEventSuccess = true;
      try {
        AuthenticationResponse sfauthResponse = apexLoginService.login();
        apexEventService.publishApexEvent(sfauthResponse, message);
      } catch (Exception exp) {
        exp.printStackTrace();
        apexEventSuccess = false;
      }

      if(apexEventSuccess) originalMessage.ack();
      else originalMessage.nack();
    };
  }
}
```

Step 3: Build, Deploy and Test the application locally or on Cloud infrastructure

```
./mvnw clean package
java -jar target/demo-0.0.1-SNAPSHOT.jar
```

Once the Code is deployed application is ready to consume messages and push it to Salesforce as and when it arrives.

6. Security Considerations

- Authentication: Use OAuth 2.0 for secure authentication.
- Data Encryption: Ensure data encryption in transit and at rest.
- Secure Storage: Ensure the JSON key file is securely store and access is limited.
- Access Control: Implement proper IAM roles and permissions for the service account.
- Data Encryption: Ensure data encryption in transit and at rest.
- Validation: Validate incoming requests to ensure they are from trusted sources.

7. Testing and Validation

- Unit Testing: Write unit tests for Apex classes to ensure functionality.
- Integration Testing: Validate end-to-end integration between Salesforce and GCP Pub/Sub.
- Send Test Messages: Use the Google Cloud Console to publish test messages to the Pub/Sub topic.
- Monitor Salesforce Logs: Check Salesforce debug logs to verify that the messages are received and processed correctly.
- Validate Data: Ensure the processed data is correctly updated or created in Salesforce as per your business logic.
- Performance Testing: Ensure the system can handle the expected message load.

8. Best Practices

- Error Handling: Implement robust error handling and retry mechanisms.
- Logging: Use Salesforce logging to monitor and troubleshoot issues.
- Scalability: Design the system to handle growth in message volume.
- Batch Processing: Implement batching to process messages in bulk in Apex.

9. Conclusion

Integrating Google Cloud Pub/Sub with Salesforce provides a powerful solution for real-time messaging and event-driven architecture. By following the steps outlined in this white paper, organizations can enhance their Salesforce applications' responsiveness, scalability, and reliability. This white paper serves as a guide for developers and architects looking to leverage the combined capabilities of Google Cloud Pub/Sub and Salesforce applications.

References

[1] Apex Developer Guide - https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_dev_guide.htm
[2] Google Pub/Sub - https://cloud.google.com/pubsub/docs
[3] Google Pub/Sub Architecture - https://cloud.google.com/pubsub/architecture
[4] Google Pub/Sub basics - https://cloud.google.com/pubsub/docs/pubsub-basics
[5] Apex Integration - https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_integration_intro.htm
[6] Named Credentials - https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_callouts_named_credentials.htm
[7] Google Pub/Sub Pull Subscription - https://cloud.google.com/pubsub/docs/create-subscription