Pub/Sub API (Pilot)

Salesforce, August 2021

@salesforcedocs

Last updated: September 7, 2021

CONTENTS

PUB/SUB API (PILOT)

Pub/Sub API Overview

The Pub/Sub API pilot provides a single interface for publishing and subscribing to platform events, including real-time event monitoring events and change data capture events. The Pub/Sub API is a gRPC API that is based on HTTP 2.

Available in: Enterprise, Performance, Unlimited, and Developer Editions

Important: This feature is not generally available and is being piloted with certain Customers subject to additional terms and conditions. It is not part of your purchased Services. This feature is subject to change, may be discontinued with no notice at any time in SFDC's sole discretion, and SFDC may never make this feature generally available. Make your purchase decisions only on the basis of generally available products and features. This feature is made available on an AS IS basis and use of this feature is at your sole risk.

The Pub/Sub API provides many benefits:

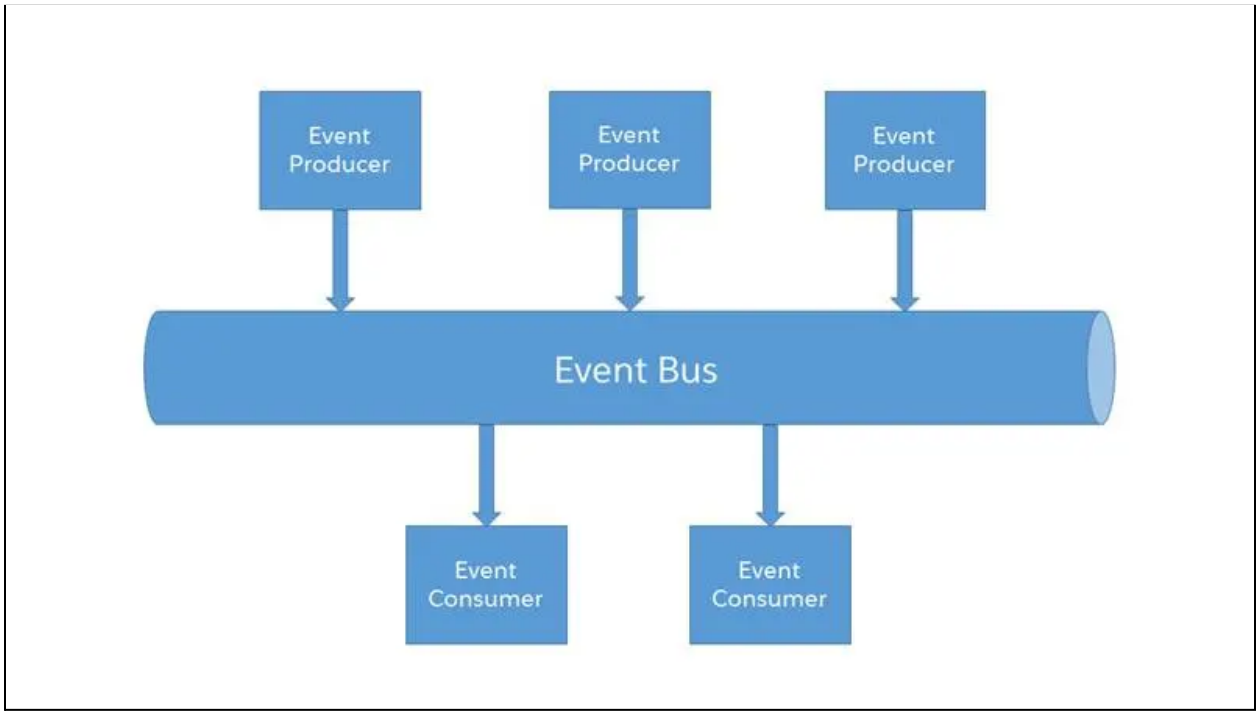
- Publishing, subscribing, and event schema retrieval all in one API.
- Guaranteed feedback for event publishing. The publish operations return the final publish results and not intermediate queueing results.
- Scalable, and secure publishing and delivery of platform events, change data capture events, and real-time event monitoring events.
- Real-time, highly performant data streaming that uses compression through HTTP 2.
- Support for 11 programming languages in the client that are offered by the gRPC API, such as Python, Java, Node, and C++. For all the supported languages, see https://grpc.io/docs/languages/.
- An active online developer community presence for gRPC.
- Bidirectional data streaming through the gRPC API. The server can deliver a stream of events to the client, and the client can publish a stream of events to the server.
- Flow control that lets developers specify how many events to receive at a time.
- Cross-cloud integration capabilities enabling the development of event-driven apps that integrate across Salesforce clouds.

The Pub/Sub API enables you to build event-driven integration apps. Here are some examples of what you can do with the Pub/Sub API.

- Subscribe to Event Monitoring real-time events and publish a platform event back into Salesforce to restrict a user's profile when they log into Salesforce after working hours.

- Subscribe to change data capture events and synchronize order data in an external inventory system.
- Subscribe to a standard platform event, such as AppointmentSchedulingEvent, and integrate with Google Calendar to update users' calendars.

Using the Pub/Sub API, you can interface with the expanded and improved Salesforce event bus by publishing and subscribing to events. The event bus is a multitenant, multicloud event storage and delivery service based on a publish-subscribe model. Platform events and change data capture events are published to the event bus, where they're stored temporarily. You can retrieve stored event messages from the event bus with the Pub/Sub API. Each event message contains the replay ID field, represented as replay_id in the protocol specification. It is an opaque ID that identifies the event in the stream and enables replaying the stream after a specific replay ID.

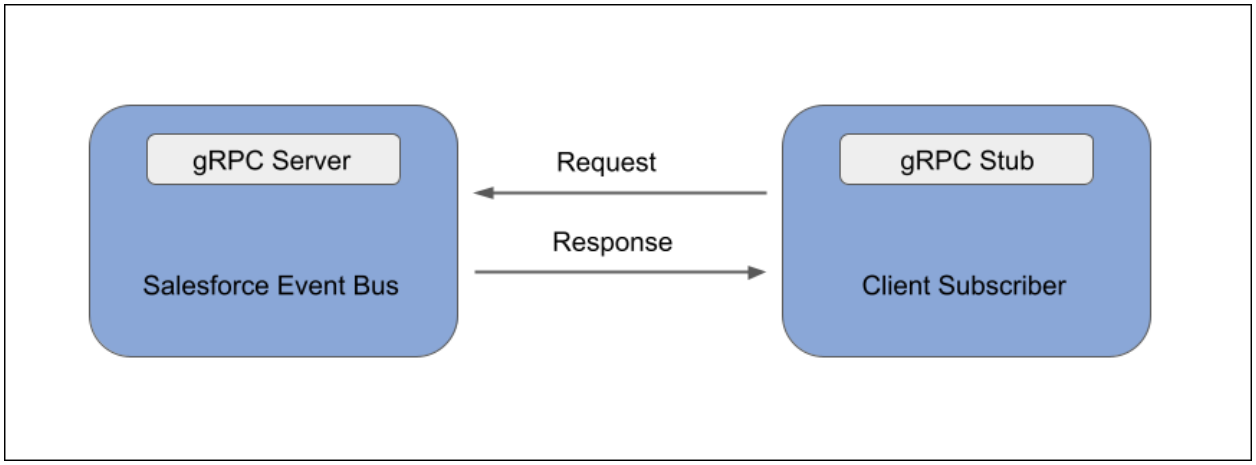


The expanded event bus service is built outside the main Salesforce CRM stack, which powers the original Salesforce products (Sales Cloud and Service Cloud). As such, the API provides cross-cloud integration capabilities between Sales and Service clouds and other Salesforce clouds, such as Marketing Cloud, Commerce Cloud, and Tableau Analytics. It also provides enhanced scalability and performance. Developers can focus on building event-driven apps that scale and that integrate across various Salesforce clouds.

Pub/Sub API as a gRPC API

Because the Pub/Sub API is a gRPC API, let's define what a gRPC API is. gRPC is an open source Remote Procedure Call (RPC) framework that enables connecting devices, mobile applications, and browsers to backend services. With gRPC, a client app can call a method on a server as if it were a local object, making it easier for you to create distributed apps and services.

gRPC requires defining a service, which specifies the methods that can be called remotely with their parameters and return types. The server implements this interface and runs a gRPC server to handle client calls. The client has a stub that mirrors the methods available on the server.



The Pub/Sub API service is defined in a proto file, with RPC method parameters and return types specified as protocol buffer messages. Our proto file example is based on the Pub/Sub API proto file but has been shortened for illustration purposes. This proto file defines the service by listing the methods to publish, subscribe, get the schema, and get the topic information.

```
message PublishRequest {
    string topic_name = 1;
    repeated ProducerEvent events = 2;
    string auth_refresh = 3;
}

message PublishResponse {
    repeated PublishResult results = 1;
    string schema_id = 2;
}

service EventBusAPIService {

    rpc Subscribe (stream FetchRequest) returns (stream FetchResponse);

    rpc GetSchema (SchemaRequest) returns (SchemaInfo);

    rpc GetTopic (TopicRequest) returns (TopicInfo);

    rpc Publish (PublishRequest) returns (PublishResponse);

    rpc PublishStream (stream PublishRequest) returns (stream PublishResponse);
}
```

The proto file lists the messages for the Publish and PublishStream methods. PublishRequest is the request message of the publish methods. PublishResponse is the response message of the publish methods. Other messages are omitted for brevity. For the full definition of the Pub/Sub API proto file, see Pub/Sub API proto file in GitHub.

Terminology

The Pub/Sub API uses these terms.

Event or Event Message

Event can refer to the event entity definition in Salesforce or the event message.

Event is a Salesforce entity that represents the definition of the data that is sent in an event message. You can define the event entity, such as with a custom platform event. Or it can be defined by Salesforce, such as a change data capture event like AccountChangeEvent.

An event message is the real-time notification that contains the data that the publisher sends and the subscriber receives. When there is no ambiguity, event is used in the documentation instead of event message for brevity.

Topic

The API name of the event object preceded by a path. The topic indicates the type of event to publish and the type of event to subscribe to. For example, the topic of a custom platform event with the API name of Order_Event__e is `/event/Order_Event__e`.

Event Bus

A multitenant, multicloud event storage and delivery service based on a publish-subscribe model. The event bus is based on a time-ordered event log, which ensures that event messages are stored and delivered in the order that they're received by Salesforce.

See Also

- *Salesforce Engineering Blog*: How Apache Kafka Inspired Our Platform Events Architecture

Generating Code from the Proto File

To generate code from the proto file, use a gRPC plug-in with protoc. This process generates client code, server code, and protocol buffer code for populating, serializing, and retrieving message types. For more information, see Introduction to gRPC in the gRPC documentation.

The quick start example walks you through the steps of generating the code from the proto file using gRPC tools for Python.

The client has a local object known as a stub that implements the service methods. When a gRPC client calls the API, the corresponding API implementation is called on the server. The gRPC infrastructure decodes incoming requests, executes service methods, and encodes service responses. The client calls the methods on the local object, wrapping the parameters for the call in the appropriate protocol buffer message type. gRPC handles sending the requests to the server and returning the server's protocol buffer responses.

Bidirectional Streaming

Bidirectional streaming is one of the four types of RPC methods that can be defined in a gRPC API. With bidirectional streaming, both the client and the server can send a stream of messages to each other. The client doesn't need to wait until the server finishes sending all the messages to send new requests. Similarly, the server doesn't need to wait until the client has sent all the messages before responding.

The Subscribe method in the example above, as well in the Pub/Sub API, uses bidirectional streaming to subscribe to an Event Bus topic. The PublishStream method also uses bidirectional streaming. For more information, see Core concepts, architecture and lifecycle in the gRPC documentation.

Event Data Serialization with Apache Avro

The event bus uses Apache Avro serialization for event payloads. As a result, your app must use Apache Avro to serialize and deserialize event payloads when sending or receiving them from the Pub/Sub API. Apache Avro is a data serialization system that provides a binary data format and a schema. Before you publish an event message, you must encode it to the Avro format. When you receive an event message, you must decode it using the Avro format before you can retrieve the contents of the event payload. For more information, see the Apache Avro Documentation.

The Python Code Example Quick Start includes example functions for encoding and decoding the event messages using Avro.

Supported Authentication

The Pub/Sub API supports any authentication mechanism that enables retrieving the session ID, including username and password authentication, and OAuth. The session ID is part of the authentication metadata header that is passed to the Pub/Sub API RPC methods. For more information about authorizing your app with OAuth, see OAuth Authorization Flows in *Salesforce Help*.

Supported Event Types

The Pub/Sub API supports high-volume platform events, including custom and standard events, real-time event monitoring events, and change data capture events. It doesn't support legacy events, such as standard-volume platform events, PushTopic events, and generic streaming events.

Python Code Quick Start Example

In this quick start, you learn how to build a Pub/Sub API client in Python. The steps walk you through generating the stub files, authenticating to Salesforce, configuring events, writing code to subscribe to events, and writing code to publish events.

Note: The steps provide enough instructions and code snippets so that you can build your own client but don't provide the full code sample. You can find full code examples in https://github.com/developerforce/pub-sub-api-pilot. However, the full examples aren't

intended for production use and haven't undergone thorough functional and performance testing. You can use these examples as a starting point to build your own client.

Step 1: Generate the Stub Files

1. Install the Python package manager by running this command in the terminal.
   `pip3 install grpc grpcio-tools avro-python3`
   You can use a different package manager or a different version of Python.
2. Clone the GitHub repository for Pub/Sub API from https://github.com/developerforce/pub-sub-api-pilot. The proto file name is `pubsub_api.proto`.
3. Generate the stubs for the Pub/Sub API by running this command.
   `python3 -m grpc_tools.protoc --proto_path=. pubsub_api.proto --python_out=. --grpc_python_out=.`
   This command generates two files in your current directory: `pubsub_api_pb2.py` and `pubsub_api_pb2_grpc.py`. It also generates client and server code, and protocol buffer code for populating, serializing, and retrieving message types.

Step 2: Build the Python Client

1. Create a Python file. For example, `PubSubAPIClient.py`.

2. Import these modules:
```
from __future__ import print_function
import grpc
import requests
import threading
import io
import pubsub_api_pb2 as pb2
import pubsub_api_pb2_grpc as pb2_grpc
import avro.schema
import avro.io
import time
import certifi
```

3. Set a semaphore at the beginning of the program. Because of the way Python gRPC is designed, the program shuts down immediately if no response comes back in the milliseconds between calling an RPC and the end of the program. By setting a semaphore, you cause the client to keep running indefinitely.

```
semaphore = threading.Semaphore(1)
```

4. Create a global variable to store the replay ID.
```
latest_replay_id = None
```

5. Set up the gRPC channel, and generate the stub. `PubSubStub` comes from the `pubsub_api_pb2_grpc.py` file, which you generated in the previous step.

```
with open(certifi.where(), 'rb') as f:
   creds = grpc.ssl_channel_credentials(f.read())
with
grpc.secure_channel('eventbusapi-core1.sfdc-ypmv18.svc.sfdcfc.net
:7443', creds) as channel:
      #All of the code in the rest of the tutorial will go inside
      # this block
```

6. Retrieve your session token. You will use the username, password, and API login URL for your Salesforce org. If you're using a production instance, the API login URL is https://login.salesforce.com/services/Soap/u/52.0/. If it's a sandbox, the URL is https://test.salesforce.com/services/Soap/u/52.0/.
Send a POST request formatted like so to the login URL:

```
username = <your username>
password = <your password>
url = <the appropriate login URL>
headers = {'content-type': 'text/xml', 'SOAPAction': 'login'}
xml = "<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>" + \
"<urn:login><urn:username><![CDATA[" + username + \
"]]></urn:username><urn:password><![CDATA[" + password + \
"]]></urn:password></urn:login></soapenv:Body></soapenv:Envelope>
"
res = requests.post(url, data=xml, headers=headers, verify=False)
#Optionally, print the content field returned
print(res.content)
```

Note: If you haven't set up a range of trusted IP addresses for your org, append a security token to your password. For more information, see Reset Your Security Token and Set Trusted IP Ranges for Your Organization.

Run this client by entering `python3 PubSubAPIclient.py` on the command line. When the request returns, you have XML-formatted data in the response content field, `res.content`. It contains a session ID wrapped within the `<sessionId>` tags. It also contains the server URL wrapped within the `<serverUrl>` tags, and the org ID within the `<organizationId>` tags under `<userInfo>`. Take note of those values because you will use them in the next step.

> **Note:** This step uses username and password authentication for simplicity, but we recommend you use OAuth in production apps. For more information, see Supported Authentication.

7. Store the authentication information (session ID, instance URL, and org ID) in a tuple called `authmetadata`. Each element in this tuple is also a tuple. You use this information when subscribing to a channel or publishing events.
    a. Replace the `sessionid` placeholder value with the session ID you got from the previous step.
    b. Replace the `instanceurl` placeholder value with the first part of the server URL you got from the previous step, without the path portion. For example, https://*MyDomainName*.my.salesforce.com.
    c. To build the tenant ID, get the subdomain. The tenant ID format is: `core/<Subdomain>/<Org ID>`
    The subdomain is one of the following:
        i. The name of My Domain if My Domain is set up. For example, it is ***MyDomainName*** in https://*MyDomainName*.my.salesforce.com. For more information, see What Is My Domain? in *Salesforce Help*.
        ii. The instance name if My Domain is not set up. For example, it is ***InstanceName*** in https://*InstanceName*.salesforce.com.
    `<Org ID> is` the org ID you got from the previous step. Alternatively, you can get your org ID value by following the steps in Find your Salesforce Organization ID in *Salesforce Help*.
    d. Replace the placeholder values for `Subdomain` and `Org ID` in the `tenantid` variable with the values you obtained.

```
sessionid = <the session ID you just got from the XML>
instanceurl = <the server URL you just got from the XML
ending in .com>
tenantid = "core/<Subdomain>/<Org ID>"
authmetadata = (('x-sfdc-api-session-token', sessionid),
        ('x-sfdc-instance-url', instanceurl),
        ('x-sfdc-tenant-id', tenantid))
```

8. Generate your stub object as follows.

```
stub = pb2_grpc.PubSubStub(channel)
```

Step 3: Set Up Events

To subscribe to change data capture events on the standard channel, select the objects for which you want to receive events.

1. In Setup, search for `Change Data Capture`, and then select **Change Data Capture**.
2. On the Change Data Capture page, select the object.
3. Click **Save**.

To subscribe to a custom channel, ensure that the custom channel is created first. After the channel is created, the entities are selected as part of the custom channel creation. See the *Change Data Capture Developer Guide*.

To subscribe to a custom platform event, define a custom platform event on the Platform Events page in Setup.

To subscribe to a standard platform event, including real-time event monitoring events, you can view the available events in the Standard Platform Event Object List in the *Platform Events Developer Guide*.

Step 4: Write Code That Subscribes to an Event Channel

1. Get the topic name that you want to subscribe to.
   The topic format is:
   - For a custom platform event: `/event/EventName__e`
   - For a standard platform event: `/event/EventName`
   - For a change data capture channel that captures events for all selected entities: `/data/ChangeEvents`
   - For a change data capture single-entity channel for a standard object: `/data/<StandardObjectName>ChangeEvent`
   - For a change data capture single-entity channel for a custom object: `/data/<CustomObjectName>__ChangeEvent`
   - For a change data capture custom channel: `/data/CustomChannelName__chn`

2. Create a generator function to make a FetchRequest stream. In this FetchRequest, `num_requested` is the maximum number of events that the server can send to the client at once. The specified `num_requested` of events can be sent in one or more FetchResponses, with each FetchResponse containing a batch of events. In this case, we set it to 1 but you can set it to how many events you're willing to process.

```
def fetchReqStream(topic):
      while True:
                  semaphore.acquire()
                  yield pb2.FetchRequest(
                      topic_name = topic,
                      replay_preset = pb2.ReplayPreset.LATEST,
                      num_requested = 1)
```

3. Create a decoding function to decode the payloads of received event messages.

```
def decode(schema, payload):
        schema = avro.schema.Parse(schema)
        buf = io.BytesIO(payload)
        decoder = avro.io.BinaryDecoder(buf)
        reader = avro.io.DatumReader(writer_schema=schema)
        ret = reader.read(decoder)
        return ret
```

4. Make the subscribe call and handle received event messages. Decode the payloads of the events with your decoding function. Store the latest replay ID received. You can use the replay ID later to restart a subscription after the last consumed event, if necessary. For more information, see Replaying an Event Stream.

```
mysubtopic = "/data/OpportunityChangeEvent"
substream = stub.Subscribe(fetchReqStream(mysubtopic),
            metadata=authmetadata)
for event in substream:
      semaphore.release()
      if event.events:
            payloadbytes = event.events[0].event.payload
            schemaid = event.events[0].event.schema_id
            schema = stub.GetSchema(
                 pb2.SchemaRequest(schema_id=schemaid),
                 metadata=authmetadata).schema_json
            decoded = decode(schema, payloadbytes)
            print("Got an event!", decoded)
      else:
            print("[", time.strftime('%b %d, %Y %l:%M%p %Z'),
            "] The subscription is active.")
      latest_replay_id = event.latest_replay_id
```

If you run your code at this point, you don't receive any event messages unless you or Salesforce publishes an event message.

For change data capture events, make a change to a Salesforce record of a supported object, such as Opportunity, so that Salesforce generates an event message. Make sure that Change Data Capture is tracking the object by checking the Change Data Capture page in Setup.

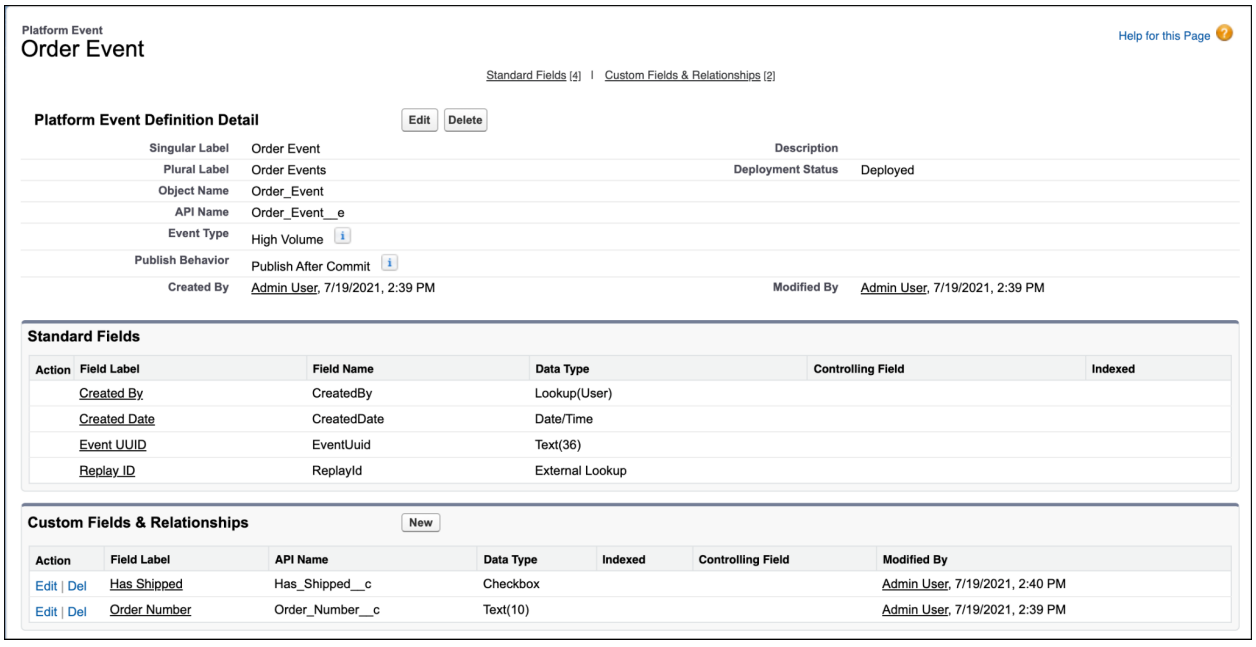
Salesforce publishes most standard platform events, including real-time event monitoring events, in response to an action in Salesforce. You can publish only the standard events that support the `create()` call. For more information, see Standard Platform Event Object List in the *Platform Events Developer Guide*.

You can also publish a custom platform event. The next step shows you how to do that using the Pub/Sub API in Python.

Step 5: Write Code That Publishes a Platform Event Message

Before you publish a custom platform event message, ensure that the platform event is defined in your org. You can view defined platform events in Setup on the Platform Events page.

For example, this image shows the definition of Order_Event__e. This event has two fields: `Order_Number__c` of type Text and `Has_Shipped__c` of type Checkbox.



Note: For these steps, we recommend creating a separate Python file for publishing so that you can run the publishing and subscribing clients independently. Include common code for creating the channel and stub, the authentication code to build `authmetadata`, and the import statements from the previous steps. Also, add this import statement: `from`

```
datetime import datetime, timedelta
```

1. Get the topic name for the event you want to publish. The topic format is /event/*EventName*__e. For example, for Order_Event__e the topic is `/event/Order_Event__e`.

2. Get the schema ID and schema for the event. To get the schema ID, call the GetTopic method and pass the topic name. Next, pass the schema ID to the GetSchema method, which returns the schema.

   ```
   mypubtopic = <your publish topic>
   schemaid = stub.GetTopic(pb2.TopicRequest(topic_name=mypubtopic),
               metadata=authmetadata).schema_id
   schema = stub.GetSchema(pb2.SchemaRequest(schema_id=schemaid),
               metadata=authmetadata).schema_json
   ```

3. Create a function to encode the information that you want to send by using the schema.

   ```
   def encode(schema, payload):
           schema = avro.schema.Parse(schema)
           buf = io.BytesIO()
           encoder = avro.io.BinaryEncoder(buf)
           writer = avro.io.DatumWriter(writer_schema=schema)
           writer.write(payload, encoder)
           return buf.getvalue()
   ```

4. Create a function that creates a PublishRequest. Construct the payload by adding the event fields and values in the payload variable. Populate the values of the required system fields: CreatedDate and CreatedById. The CreatedById value isn't validated. For `<event field>: <field value>`, list the event fields and values. For example, for Order_Event__e, you can add:
   `"Order_Number__c": "100",`
   `"Has_Shipped__c": True`

   The req variable contains the encoded payload, which is returned by the encode function. It also contains the schema ID. The id field uniquely identifies the event message and helps correlate the published event message with the received one. Ideally, assign a UUID value to this field. However, you can also supply an arbitrary string value, like in this example.

   ```
   def makePublishRequest(schemaid):
       dt = datetime.now() + timedelta(days=5)
       payload = {
           "CreatedDate": int(datetime.now().timestamp()),
           "CreatedById": '005R...', #Your user ID
   ```

```
        <event field>: <field value>
    }
    req = {
        "id": "234", # Event ID
        "schema_id": schemaid,
        "payload": encode(schema, payload)
    }

    return [req]
```

5. Make the publish call and handle any acknowledgements you get back:

   ```
   publishresponse =
   stub.Publish(pb2.PublishRequest(topic_name=mypubtopic, events=
   makePublishRequest(schemaid)), metadata=authmetadata)
   ```

   If the publish request is successful, you receive a PublishResponse message containing the replay ID.

   If the publish request was not successful, you get an error back similar to the following:

   ```
   results {
     error {
       code: PUBLISH
       msg: "com.salesforce.eventbus.exceptions.PublishException:
   Unsupported topic [/event/Tracker_Event__e]. Standard Volume
   event type is not supported."
     }
   }
   schema_id: "AKTsT5i0mDe_UF8qnC8Aig"
   ```

Pub/Sub API Proto File

You can get the Pub/Sub API proto file from the pub-sub-api-pilot GitHub repository.
For information about the protocol buffer language, see Language Guide (proto 3).

RPC Methods in the Pub/Sub API

Subscribe RPC Method

The Subscribe method uses bidirectional streaming. It is pull-based, which means that it requests new events. This model is in contrast to push-based subscription in which the subscriber is a listener that waits for events to be sent.

```
rpc Subscribe (stream FetchRequest) returns (stream FetchResponse);
```

A subscriber can request more events as it consumes events. This behavior enables a client to handle flow control. The typical flow is:

1. Client requests X number of events via FetchRequest.
2. Server receives the request and delivers events until X events are delivered to the client via one or more FetchResponses.
3. Client consumes the FetchResponses as they're received.
4. Client issues a new FetchRequest for Y number of events. This request can come before the server has delivered the earlier X number of events so that the client gets a continuous stream of events, if any.

If a client requests more events before the server finishes the last requested amount, the server appends the new amount to the current batch of events it still needs to fetch and deliver.

Keepalive Behavior

If there are no events to deliver, the server sends an empty batch FetchResponse with the latest replay_id. The empty FetchResponse is sent within 270 seconds. An empty FetchResponse provides a periodic keepalive from the server, which indicates that the subscription is alive, and the latest replay_id. If a client has to resubscribe, the replay_id enables it to restart the subscription after the last consumed event and not have to replay an old stream of events. Save the bytes of the last received replay_id so that you can supply it in the new subscribe call. For more information, see Replaying an Event Stream.

Replaying an Event Stream

A client can subscribe at any position in the stream by providing a replay option in the first FetchRequest. Any subsequent FetchRequests with a new replay option are ignored. A client needs to call the Subscribe RPC again to restart the subscription at a new position in the stream. The replay option consists of a combination of replay_preset and replay_id values in the first FetchRequest received from a client.

- To subscribe from the tip of the stream, specify a replay_preset of LATEST.
- To resubscribe after a specific replay_id, in the first FetchRequest, set replay_preset to CUSTOM. Also, set the replay_id to the replay_id of the last keepalive message or the last processed event message, whichever was received last.
- To subscribe from the earliest retained events, specify a replay_preset of EARLIEST. Use EARLIEST sparingly because it can slow performance when retrieving a large number of events.
- If no replay_preset is specified in the FetchRequest, the subscription starts at the tip of the stream (LATEST).

The first FetchRequest of the stream identifies the topic to subscribe to. If a subsequent FetchRequest provides topic_name, it must match what was provided in the first FetchRequest. Otherwise, the RPC sends an error with an INVALID_ARGUMENT status.

For more information about the fields in FetchRequest and FetchResponse, see the Pub/Sub API proto file.

Publish RPC Method

Two publish methods are defined in the Pub/Sub API service: Publish and PublishStream.

The Publish method is a unary RPC, which means that it sends only one request and receives only one response. It synchronously publishes the batch of events in PublishRequest to an Event Bus topic. After publishing the event messages, the server sends back a response to the client. Use Publish if you want to know the status of a publish operation before publishing the next batch of event messages.

```
rpc Publish (PublishRequest) returns (PublishResponse);
```

The PublishResponse holds a PublishResult for each event published that indicates success or failure of the publish operation. A successful status means that the event was published. A failed status means that the event failed to publish, and the client can retry publishing this event.

PublishStream RPC Method

The PublishStream method uses bidirectional streaming. It can send a stream of publish requests while receiving a stream of publish responses from the server.

```
rpc PublishStream (stream PublishRequest) returns (stream
PublishResponse);
```

The first PublishRequest of the stream identifies the topic to publish on. If a subsequent PublishRequest provides topic_name, it must match what was provided in the first PublishRequest. Otherwise, the RPC sends an error with an INVALID_ARGUMENT status.

The server returns a PublishResponse for each PublishRequest when publishing is complete for the batch. A client doesn't have to wait for a PublishResponse before sending a new PublishRequest. Multiple publish batches can be queued up. This behavior allows for a higher publish rate, because a client can asynchronously publish more events while publishes are still in flight on the server side.

The PublishResponse holds a PublishResult for each event published that indicates success or failure of the publish operation. A successful status means that the event was published. A failed status means that the event failed to publish, and the client can retry publishing this event.

To hold onto the stream, a client must send a valid publish request with one or more events every 70 seconds. Otherwise, the server closes the stream and notifies the client. When the client is notified of the stream closure, it must make a new `PublishStream` call to resume publishing.

For more information about the fields in PublishRequest and PublishResponse, see the Pub/Sub API proto file.

GetSchema RPC Method

The GetSchema method returns the schema of an event topic using the schema ID. Use the schema to encode the payload in the Avro format of the event to publish, or to decode the payload of a received event.

Because the schema typically doesn't change often, we recommend you call GetSchema once and use the returned schema for all operations. If the event schema changes, for example, when an administrator adds a field to the event definition, the schema ID changes. We recommend you store the schema ID and compare it with the latest schema ID retrieved from PublishResponse or FetchResponse. If the schema ID changes, call GetSchema to retrieve the new schema.

```
rpc GetSchema (SchemaRequest) returns (SchemaInfo);
```

To get the schema ID for the `SchemaRequest` parameter, do one of the following:

- Call GetTopic. The return value of this method is TopicInfo. TopicInfo contains schema_id, which represents the latest schema. We recommend you publish events with the latest schema.
- For events received from the event bus, get the schema ID from the event message in the FetchResponse, ProducerEvent.schema_id. Use this schema for deserialization. For events published to the event bus, get the schema ID from the PublishResponse.

Note: You can still publish events with an old schema saved from an earlier GetTopic call. You use an Avro code generator to generate classes based on Avro types and deploy your app. If the event schema changes later, you can still publish and subscribe to the events as long as the schema differences are resolvable by the Avro schema resolution rules.

For more information about the fields in SchemaRequest and SchemaInfo, see the Pub/Sub API proto file.

GetTopic RPC Method

Returns information for an event, such as the event topic name and the schema ID. The schema ID is used to get the event schema with `GetSchema`.

```
rpc GetTopic (TopicRequest) returns (TopicInfo);
```

For more information about the fields in TopicRequest and TopicInfo, see the Pub/Sub API proto file.

Handling Errors

If an error occurs while an RPC method executes, the Pub/Sub API throws a gRPC StatusRuntimeException that contains a status code. The gRPC status codes can be found here. In your code, catch the exceptions after performing an RPC method call and handle the error. After catching the exception, you can call the getStatus() method on the exception to get the Status. Also, the Pub/Sub API adds a custom error code in the Trailers section of the exception. You can retrieve the custom error code by calling getTrailers() on the exception.

Exception Example

In the example gRPC exception below, the status returned is `PERMISSION_DENIED`. The custom error code from the Pub/Sub API is:

```
[Trailer] = error-code [Value] =
sfdc.platform.eventbus.grpc.topic.meta.permission

 === GRPC Exception ===
io.grpc.StatusRuntimeException: PERMISSION_DENIED: Error getting metadata for
tenant core/flash232cdpusercom/00DR0000000KvItMAK for topic
/event/DataStreamStatusEvent__e. Possible reason would be incorrect topic name
or incorrect credentials.
 === Trailers ===
[Trailer] = internal-cause [Value] =
com.salesforce.eventbus.error.EBAPIResourceException:
com.salesforce.eventbus.exceptions.MetadataException: 501::Exception
retrieving metadata with error: Invalid topic (entity not found):
/event/DataStreamStatusEvent__e  at
com.salesforce.eventbus.api.retry.TopicRetryable.proceedWithException(TopicRet
ryable.java:53)  at
com.salesforce.eventbus.api.retry.TopicRetryable.lambda$getTopicRetry$0(TopicR
etryable.java:38)  at
java.base/java.util.concurrent.CompletableFuture.uniWhenComplete(CompletableFu
ture.java:859)  at
java.base/java.util.concurrent.CompletableFuture$UniWhenComplete.tryFire(Compl
etableFuture.java:837)  at
java.base/java.util.concurrent.CompletableFuture$Completion.exec(CompletableFu
```

```
ture.java:479)  at
java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:290)  at
java.base/java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec(ForkJoinPoo
l.java:1020)  at
java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1656)  at
java.base/java.util.concurre
[Trailer] = content-type [Value] = application/grpc
[Trailer] = span-id [Value] = spanId
[Trailer] = error-code [Value] =
sfdc.platform.eventbus.grpc.topic.meta.permission
[Trailer] = type [Value] = GetTopic
```

Error Codes

This table lists the error codes that the Pub/Sub API returns as part of the gRPC StatusRuntimeException. The gRPC status codes can be found in the gRPC documentation.

Error Code	Error Description
General errors	
`sfdc.platform.eventbus.grpc.service.unavailable`	The Pub/Sub API service is unavailable.
`sfdc.platform.eventbus.grpc.service.auth.error`	An authentication exception occurred. Provide valid authentication via metadata headers.
`sfdc.platform.eventbus.grpc.service.auth.headers.invalid`	The auth headers value for the specified key is invalid. Provide valid auth headers. The auth headers can't be blank.
`sfdc.platform.eventbus.grpc.service.auth.refresh.invalid`	The auth refresh token value is invalid. Provide a valid auth refresh value.
`sfdc.platform.eventbus.grpc.service.protection.triggered`	The service received too many connections and doesn't have the resources to accept new connections.
`sfdc.platform.eventbus.grpc.service.tenant.license`	The Salesforce org is not licensed to access the Pub/Sub API. Contact Salesforce to enable the API.
Schema errors	
`sfdc.platform.eventbus.grpc.schema.meta.permission`	An error occurred while getting the schema. Possible reasons are an incorrect schema ID or incorrect credentials.

`sfdc.platform.eventbus.grpc.schema.api.unavailable`	The schema information is unavailable.
`sfdc.platform.eventbus.grpc.schema.validation.failed`	Schema validation failed. The schema ID can't be blank.
Topic errors	
`sfdc.platform.eventbus.grpc.topic.meta.permission`	An error occurred while getting the metadata for the specified topic. Possible reasons are an incorrect topic name or incorrect credentials.
`sfdc.platform.eventbus.grpc.topic.api.unavailable`	The topic information is unavailable for the specified topic.
`sfdc.platform.eventbus.grpc.topic.validation.empty`	An error occurred while validating the topic information provided. The topic can't be blank.
Publish errors	
`sfdc.platform.eventbus.grpc.publish.event.count.invalid`	The request contains no events.
`sfdc.platform.eventbus.grpc.publish.topic.mismatch`	There is a mismatch of topic names between requests.
`sfdc.platform.eventbus.grpc.publish.auth.refresh.invalid`	The refresh token shouldn't be provided in the initial request.
`sfdc.platform.eventbus.grpc.publish.topic.validation.empty`	An error occurred while validating the provided topic information. The topic can't be blank.
`sfdc.platform.eventbus.grpc.publish.stream.sweeper.timeout`	No publish request was received during the timeout period of 120 seconds. The publish stream timed out.
Subscription errors	
`sfdc.platform.eventbus.grpc.subscription.fetch.requested.events.invalid`	The requested number of events in a fetch request must be greater than zero.
`sfdc.platform.eventbus.grpc.subscription.fetch.topic.mismatch`	There is a mismatch of topic names between requests.

`sfdc.platform.eventbus.grpc.subscription.fetch.replayid.validation.failed`	The Replay ID validation failed. Provide a Replay ID in the custom preset.
`sfdc.platform.eventbus.grpc.subscription.fetch.replayid.corrupted`	The Replay ID validation failed. Ensure that the Replay ID is valid.
`sfdc.platform.eventbus.grpc.subscription.topic.cannot.subscribe`	Can't subscribe to the specified topic. Check that you have the required permissions.
`sfdc.platform.eventbus.grpc.subscription.fetch.replay.repeated`	Due to an internal error, the server received an event that is older than the one received earlier. Its Replay ID value is lower than that of the last received event.
`sfdc.platform.eventbus.grpc.subscription.fetch.overflow`	Too many total events were requested for this subscription. Process some events first before sending a new fetch request.
`sfdc.platform.eventbus.grpc.subscription.internal.error`	The subscription encountered an internal error and can't continue. Try restarting the subscription.

Event Allocations

Check out allocations for the event message size, how many events you can publish, and how many events can be delivered to subscribers.

Event Message Size

For optimal performance, we recommend that the event message size in a publish request doesn’t exceed 3 MB. The maximum event message size is 4 MB, which gRPC enforces.

If you exceed the event message size of 4 MB in a publish request, you get a gRPC error with this status.

```
Status{code=CANCELLED, description=RST_STREAM closed stream. HTTP/2 error code: CANCEL, cause=null}
```

Event Publishing Allocation

We recommend you send no more than 200 events in one publish request for best performance results.

Platform events are subject to an hourly publishing allocation. For more information, see Platform Event Allocations in the *Platform Events Developer Guide.*

Event Delivery Allocations

When you subscribe with the Pub/Sub API, the event delivery allocations for platform events, change data capture events, and real-time event monitoring events apply. For more information, see the documentation for each event type.

- *Platform Events Developer Guide*: Platform Event Allocations
- *Change Data Capture Developer Guide*: Change Data Capture Allocations
- *Salesforce Help*: Real-Time Event Monitoring Data Streaming