



Pub/Sub API (Pilot)

Salesforce,
August–November 2021



[@salesforcedocs](https://twitter.com/salesforcedocs)

Last updated: April 26, 2022

© Copyright 2000–2022 Salesforce, Inc. All rights reserved. Salesforce is a registered trademark of Salesforce, Inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

| | |
|--------------------------------------------------------------|-----------|
| PUB/SUB API (PILOT) | 5 |
| Pub/Sub API (Pilot) Product and Documentation Updates | 5 |
| Pub/Sub API Overview | 8 |
| Pub/Sub API as a gRPC API | 9 |
| Terminology | 11 |
| Event or Event Message | 11 |
| Topic | 12 |
| Event Bus | 12 |
| Generating Code from the Proto File | 12 |
| Bidirectional Streaming | 12 |
| Event Data Serialization with Apache Avro | 13 |
| Event Deserialization Considerations | 13 |
| Bitmap Fields in Change Events | 13 |
| Date Fields | 14 |
| Change Event Differences with Streaming API (CometD) | 14 |
| Supported Authentication | 15 |
| Supported Event Types | 15 |
| Python Code Quick Start Example | 15 |
| Step 1: Generate the Stub Files | 15 |
| Step 2: Build the Python Client | 16 |
| Step 3: Set Up Events | 18 |
| Step 4: Write Code That Subscribes to an Event Channel | 18 |
| Received Event Example | 20 |
| Step 5: Write Code That Publishes a Platform Event Message | 22 |
| Pub/Sub API Proto File | 24 |
| Other Code Examples | 24 |
| Go Code Examples | 24 |
| RPC Methods in the Pub/Sub API | 25 |
| Subscribe RPC Method | 25 |
| Keepalive Behavior | 25 |
| Replaying an Event Stream | 25 |
| Publish RPC Method | 26 |
| PublishStream RPC Method | 27 |
| GetSchema RPC Method | 28 |
| GetTopic RPC Method | 28 |

| | |
|-------------------------------------|-----------|
| Headers for RPC Method Calls | 29 |
| Handling Errors | 30 |
| Exception Example | 31 |
| Error Codes | 31 |
| Event Allocations | 34 |
| Event Message Size | 34 |
| Event Publishing Allocation | 34 |
| Event Delivery Allocations | 35 |

PUB/SUB API (PILOT)

Pub/Sub API (Pilot) Product and Documentation Updates


| Date | Changes |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 4/26/2022 | <p>Product and documentation changes:</p> <ul style="list-style-type: none">• The gRPC exception that Pub/Sub API throws no longer contains the <code>internal-cause</code> trailer. This trailer was for internal use only. This trailer was removed from the exception example in this guide. (See Exception Example.) |
| 3/29/2022 | <p>Code sample changes:</p> <ul style="list-style-type: none">• We added new code examples for the Go programming language to the Pub/Sub API GitHub repo in a new folder named <code>go</code>.• The existing Python code examples were moved under the <code>python</code> folder.• The Pub/Sub API proto file has been updated to support the Go examples.• Other updates made to the Pub/Sub API proto file include adding the <code>rpc_id</code> field in <code>PublishResponse</code> (see Handling Errors) and using the new names for session authentication headers that were introduced earlier (see Headers for RPC Method Calls). <p>Documentation changes:</p> <ul style="list-style-type: none">• Added a link to the new Go examples and updated existing links so that they work in the restructured GitHub repo. (See Other Code Examples.)• Updated the Python quick start example so that it runs in the new directory structure. (See Python Code Quick Start Example.) |
| 3/17/2022 | <p>Product and documentation changes:</p> <ul style="list-style-type: none">• Documented the new RPC ID returned in <code>StatusRuntimeException</code> and updated the exception example. The RPC ID identifies the method execution that caused the exception. (See Handling Errors.)• Documented the new <code>sfdc.platform.eventbus.grpc.topic.not.found</code> error. (See Error Codes.) |

| | |
|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | <ul style="list-style-type: none"> Updated the description of the <code>sfdc.platform.eventbus.grpc.service.tenant.license</code> error. (See Error Codes.) |
| 10/29/2021 | <p>Documentation enhancements:</p> <ul style="list-style-type: none"> Added a link to a code example for decoding bitmap fields for change data capture events. (See Bitmap Fields in Change Events.) Clarified the description of the <code>instanceurl</code> header to include the various types of Salesforce URLs and authentication methods. (See Headers for RPC Method Calls.) Corrected the PublishRequest size limit to indicate that it's the limit of the batch of events. Added the limit of an individual event. (See Event Message Size.) |
| 10/6/2021 | <p>Product and documentation changes:</p> <ul style="list-style-type: none"> New Pub/Sub API endpoint: <code>api.pilot.pubsub.salesforce.com</code>. The old endpoint, <code>eventbusapi-core1.sfdc-ypmv18.svc.sfdcfc.net</code>, is still supported. (See Step 2: Build the Python Client.) RPC method headers have been shortened. The original header names are still supported. (See Headers for RPC Method Calls and Step 2: Build the Python Client.) The tenant ID has been simplified. You can supply the Salesforce org ID as the tenant ID value. The API constructs the remaining values for the tenant ID. (See Headers for RPC Method Calls and Step 2: Build the Python Client.) |
| 10/01/2021 | <p>Documentation enhancements:</p> <ul style="list-style-type: none"> Added information about the Avro binary format that the Pub/Sub API uses and event deserialization considerations. (See Event Data Serialization with Apache Avro and Event Deserialization Considerations.) Added use cases for when to use replay options for the Subscribe method. (See Replaying an Event Stream.) Clarified the value of the <code>replay_id</code> returned in an empty FetchResponse. (See Keepalive Behavior.) |
| 9/7/2021 | <p>Documentation enhancements:</p> <ul style="list-style-type: none"> Fixed an issue with the quick start code sample for setting up the gRPC channel. (See Step 2: Build the Python Client.) |
| 8/30/2021 | <ul style="list-style-type: none"> Initial pilot release. |

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.

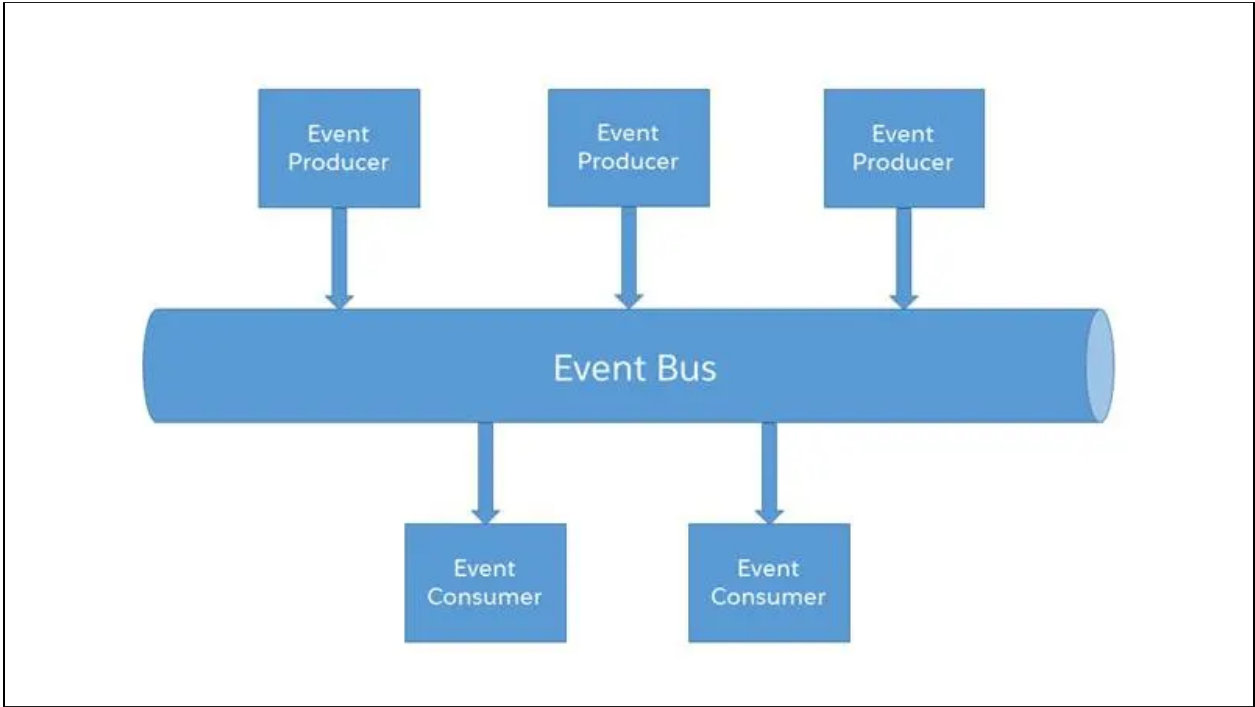
The Pub/Sub API provides many benefits:

- Publishing, subscribing, and event schema retrieval all in one API.
- Final publish results of publish operations, 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 client and the server can send a sequence of messages to each other using two independent streams.
- 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.

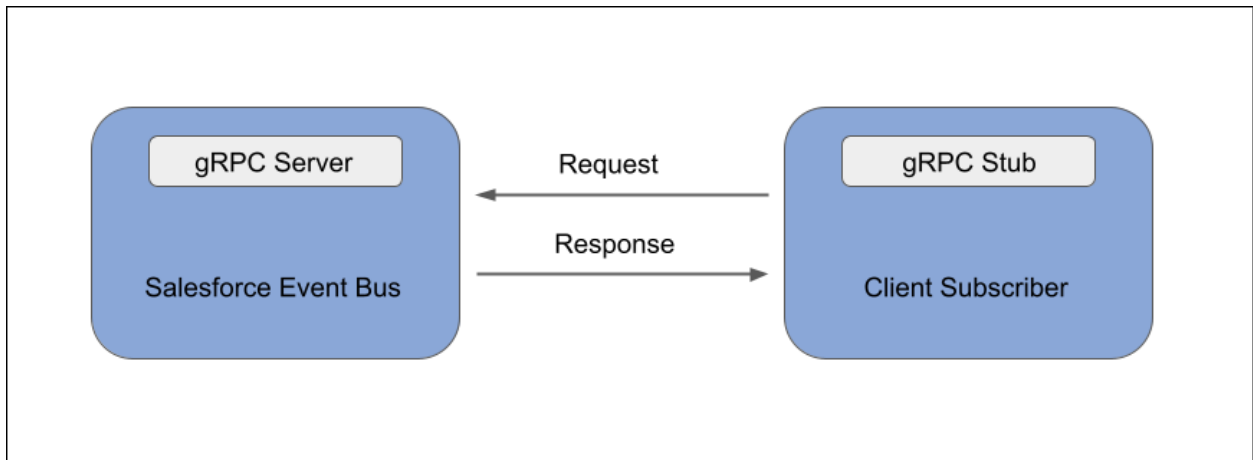


The expanded event bus service is built outside the main Salesforce CRM stack, which powers the original Salesforce products (Sales Cloud and Service Cloud). As such, the API provides cross-cloud integration capabilities between Sales and Service clouds and other Salesforce clouds, such as Marketing 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 {
  // Topic to publish on
  string topic_name = 1;
  // Batch of ProducerEvent(s) to send
  repeated ProducerEvent events = 2;
  // Authentication refresh token if applicable
  string auth_refresh = 3;
}

message PublishResponse {
  // Publish results
  repeated PublishResult results = 1;
  // Schema fingerprint for this event which is a hash of the schema
  string schema_id = 2;
  // RPC Id to trace errors. This will only be populated if publish fails.
  string rpc_id = 3;
}

service PubSub {
  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 sequence of

messages to each other using two independent streams. 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 Pub/Sub API enables the publishing and delivery of binary-encoded events using the Apache Avro schema. Apache Avro is a data serialization system that provides a binary or JSON data encoding and a schema. The Avro binary encoding is more efficient than the Avro JSON encoding because it enables faster serialization and produces smaller sizes of serialized data. For more information, see [Data Serialization and Deserialization](#) in the Apache Avro specification.

When sending or receiving events from the Pub/Sub API, your app must use Apache Avro to serialize and deserialize event payloads. 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.

Event Deserialization Considerations

The Pub/Sub API delivers events in the raw Avro binary format without modifying the fields in the event message. As a result, some fields in the deserialized event aren't readable when you print them for debugging and must be decoded before they can be processed in the subscriber. Also, the fields received in change data capture events with Pub/Sub API and Streaming API (CometD) have differences.

Bitmap Fields in Change Events

Change data capture events contain bitmap fields whose contents aren't readable when printed and must be decoded for processing in the subscriber app logic. The Pub/Sub API delivers the events in the raw Avro binary format without converting these fields. The bitmap fields are:

- `changedFields`—Contains the fields that were changed.
- `diffFields`—Contains the fields whose values were sent as a data diff.
- `nullFields`—Contains the fields that were set to null.

When you print these fields, you get a hexadecimal value and not the field names. For example:
'changedFields': ['0x650004E0']

A bitmap field uses a bitmap, encoded as a hexadecimal string, to represent the fields. This method is more space efficient than using a list of field names. A bit set to 1 indicates that the field at the corresponding position in the Avro schema was changed for `changedFields`, for example.

A bitmap field is an array of strings. The first element of the array contains a bitmap for the individual fields. Compound fields are placed in additional elements of the array with bitmaps indicating nested fields. The format for the additional array elements is “<ParentFieldPosition>-<NestedFieldBitmap>”.

To decode the bitmap, match it against the fields in the schema. The GitHub repository contains a code example that shows how to decode bitmap fields in Python. See `ChangeEventHeaderUtility.py` in [Pub/Sub API Examples - Utility Code](#). In the `SalesforceListener.py` example, the `process_confirmation` function shows how to get and convert the `changedFields` bitmap field by calling the `process_bitmap` function.

Date Fields

After you deserialize a received platform event or change event in your client, Date and Date/Time fields are in Epoch time. As a result, when you print them, you get a number. For example:

```
'CreatedDate': 1632858587281
```

If you want to make the Date field readable for debugging purposes, convert the Epoch format into another date format. For more information, see [Unix time](#).

Change Event Differences with Streaming API (CometD)

Change data capture events that are delivered with the Pub/Sub API contain some fields that aren't present on the change events received with Streaming API (CometD). This difference is because Streaming API uses JSON encoding for delivered events while Pub/Sub API uses the Avro binary format.

- Change events received with Pub/Sub API contain all the record fields, including the unchanged fields. Unchanged fields have an empty value in the change event, for example, `'Description': None`, even if they have a value in the Salesforce record. The `changedFields` field indicates which fields have changed. The `nulledFields` field indicates which fields were set to null. In contrast, change events received with Streaming API (CometD) contain only the changed fields and exclude unchanged fields.
- Change events received with Pub/Sub API contain header fields that aren't included in Streaming API events. The header fields are: `diffFields` and `nulledFields`. These fields are present on events received in Apex triggers. For more information, see [Change Event Triggers](#) in the *Change Data Capture Developer Guide*.

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 grpcio 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. Switch to the python directory.

```
cd python
```
4. Generate the stubs for the Pub/Sub API by running this command from the cloned directory.

```
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('api.pilot.pubsub.salesforce.com: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 that you got from the previous step.
 - b. Replace the `instanceurl` placeholder value with the first part of the server URL that you got from the previous step, without the path portion. For example, <https://MyDomainName.my.salesforce.com>.

- c. Replace the `tenantid` placeholder value with the org ID that 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*. For more information about the headers, see [Headers for RPC Method Calls](#).

```
sessionid = <the session ID you just got from the XML>
instanceurl = <the server URL you just got from the XML
ending in .com>
tenantid = <org ID>
authmetadata = (('access_token', sessionid),
               ('instanceurl', instanceurl),
               ('tenantid', 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(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](#).

Received Event Example

If you subscribe to the `/data/OpportunityChangeEvent` topic and you make a change to an opportunity, you receive a change event similar to this event message. This event message is for an opportunity whose Amount field was changed and the Type field was cleared (set to null).

```

{
  "ChangeEventHeader": {
    "entityName": "Opportunity",
    "recordIds": [
      "006T1000001rD83IAE"
    ],
    "changeType": "UPDATE",
    "changeOrigin": "",
    "transactionKey": "000035ad-9381-f683-046a-17395e189e78",
    "sequenceNumber": 1,
    "commitTimestamp": 1632776691000,
    "commitNumber": 68582833422,
    "commitUser": "005T1000000HjlbIAC",
    "nulledFields": [
      "0x0800"
    ]
  }
}

```

```

    ],
    "diffFields": [
    ],
    "changedFields": [
        "0x21000840"
    ]
},
"AccountId": "None",
"IsPrivate": "None",
"Name": "None",
"Description": "None",
"StageName": "None",
"Amount": 200000.0,
"Probability": "None",
"ExpectedRevenue": "None",
"TotalOpportunityQuantity": "None",
"CloseDate": "None",
"Type": "None",
"NextStep": "None",
"LeadSource": "None",
"IsClosed": "None",
"IsWon": "None",
"ForecastCategory": "None",
"ForecastCategoryName": "None",
"CampaignId": "None",
"HasOpportunityLineItem": "None",
"Pricebook2Id": "None",
"OwnerId": "None",
"CreatedDate": "None",
"CreatedBy": "None",
"LastModifiedDate": 1632776690000,
"LastModifiedById": "None",
"LastStageChangeDate": "None",
"ContactId": "None",
"ContractId": "None",
"LastAmountChangedHistoryId": "008T1000003821TIAQ",
"LastCloseDateChangedHistoryId": "None"
}

```

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.

The screenshot shows the Salesforce Platform Event configuration page for 'Order Event'. It includes sections for 'Platform Event Definition Detail', 'Standard Fields', and 'Custom Fields & Relationships'.

Platform Event Definition Detail

| | | | |
|------------------|--------------------------------|-------------------|--------------------------------|
| Singular Label | Order Event | Description | |
| Plural Label | Order Events | Deployment Status | Deployed |
| Object Name | Order_Event | | |
| API Name | Order_Event__e | | |
| Event Type | High Volume | | |
| Publish Behavior | Publish After Commit | | |
| Created By | Admin User, 7/19/2021, 2:39 PM | Modified By | Admin User, 7/19/2021, 2:39 PM |

Standard Fields

| Action | Field Label | Field Name | Data Type | Controlling Field | Indexed |
|--------|--------------|-------------|-----------------|-------------------|---------|
| | Created By | CreatedBy | Lookup(User) | | |
| | Created Date | CreatedDate | Date/Time | | |
| | Event UUID | EventUuid | Text(36) | | |
| | Replay ID | ReplayId | External Lookup | | |

Custom Fields & Relationships

| Action | Field Label | API Name | Data Type | Indexed | Controlling Field | Modified By |
|------------|--------------|-----------------|-----------|---------|-------------------|--------------------------------|
| Edit Del | Has Shipped | Has_Shipped__c | Checkbox | | | Admin User, 7/19/2021, 2:40 PM |
| Edit Del | Order Number | Order_Number__c | Text(10) | | | Admin User, 7/19/2021, 2:39 PM |



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(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 \(proto 3\)](#).

Other Code Examples

The [Pub/Sub API GitHub repository](#) contains code examples in other programming languages so you can learn how to use Pub/Sub API in those languages.

Go Code Examples

The [Go code examples](#) are a collection of code examples in the Go programming language for each RPC method call. You can use these examples as a learning resource to learn how to implement Pub/Sub API in Go.

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` as a periodic keepalive message. The empty FetchResponse is sent within 270 seconds to indicate that the subscription is alive. The returned `replay_id` can be of the last event consumed in your client or an advanced position in the stream beyond the last consumed event. An advanced position in the stream is possible even if you didn't receive events because the event bus combines streams from multiple orgs. No org-specific information or personally identifiable information is shared with other orgs.

For best performance results, we recommend that you save the `replay_id` from each empty FetchResponse and use it when resubscribing. That way, you restart the subscription to get the next unprocessed event that is stored in the event bus, and you don't have to fetch an old stream of events.

Replaying an Event Stream

Platform events and change data capture events are retained in the event bus for 3 days. 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.

This table describes the replay options and when to use each.

| <code>replay_preset</code> | Behavior | When to Use |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LATEST | Default if no replay preset is specified. Subscribe from the tip of the stream. | Use when you're interested only in new event messages and don't need the earlier event messages stored in the event bus. |
| CUSTOM | Resubscribe after a specific <code>replay_id</code> . To use this option, also set the <code>replay_id</code> to the replay ID of the last keepalive message or the last processed event message, whichever was received last. | Use to catch up on missed events after a certain event message, for example, after a connection failure. |
| EARLIEST | Subscribe from the earliest retained events. | Use to catch up on missed events and retrieve all stored events. Use this option after a client has been disconnected for more than 3 days and the last saved replay ID is no longer valid. Use EARLIEST sparingly because it can slow performance when retrieving a large number of events. |

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](#).

Headers for RPC Method Calls

Every time an RPC method executes, it uses the headers to authorize access to resources.

How you set the headers depends on your programming language. In Python, the headers are provided in a tuple as a parameter to each call. For more information, see [Python Code Quick Start Example](#). In other languages, such as Java, you can provide the headers in the stub separately before making a gRPC call.

| Header Name | Description |
|--------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>accesstoken or x-sfdc-api-session-token</p> | <p>The OAuth access token or the session ID returned in the <code>login()</code> call response.</p> <p>For more information about getting an OAuth access token, see OAuth Authorization Flows in <i>Salesforce Help</i>. For more information about the <code>login()</code> call, see login() in the <i>SOAP API Developer Guide</i>.</p> |
| <p>instanceurl or x-sfdc-instance-url</p> | <p>The first part of your Salesforce server URL without the ending path portion. You can get this URL using one of these ways.</p> <ul style="list-style-type: none"> From the <code>login()</code> call result in <code>serverUrl</code>, as outlined in the quick start example in this guide. For more information, see LoginResult in the <i>SOAP API Developer Guide</i>. From the result that Salesforce returns as part of the OAuth authorization flow in <code>instance_url</code>. For more information, see OAuth 2.0 User-Agent Flow for Desktop or Mobile App Integration in <i>Salesforce Help</i>. <p>The Salesforce URL can be:</p> <ul style="list-style-type: none"> A MyDomain URL. For example, <code>https://MyDomainName.my.salesforce.com</code>. A custom domain URL. For example, <code>https://www.example.com</code>. An instance URL if MyDomain or a custom domain is not set up. For |

| | |
|-----------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | <p>example, https://InstanceName.salesforce.com.</p> |
| <p>tenantid or x-sfdc-tenant-id</p> | <p>An ID that uniquely identifies your org. If you set this header to the org ID, the Pub/Sub API constructs the entire value for you. These values are valid:</p> <ul style="list-style-type: none"> • <Org ID> • core/<Subdomain>/<Org ID> <p>Examples:</p> <ul style="list-style-type: none"> • Org ID: 00D5e000003TTrB • Entire value: core/MyDomainName/00D5e000003TTrB <p>The org ID is returned in the login() call response. Alternatively, you can get your org ID value by following the steps in Find your Salesforce Organization ID in <i>Salesforce Help</i>.</p> <p>If you build the entire value, get the subdomain. The subdomain is one of the following:</p> <ul style="list-style-type: none"> • If My Domain is set up, the name of My Domain. For example, it's MyDomainName in https://MyDomainName.my.salesforce.com. For more information, see What Is My Domain? in <i>Salesforce Help</i>. • If My Domain is not set up, the instance name. For example, it's InstanceName in https://InstanceName.salesforce.com. |

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](#).

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. The error code provides information about the cause of the failure. For more information, see [Error Codes](#).

Also, the exception contains an RPC ID that the Pub/Sub API appends to the error message after the "rpcId:" prefix. The RPC ID is also included in the Trailers section of the exception. The RPC ID identifies the method execution that caused the exception and can aid Salesforce Customer Support in troubleshooting the error. If you can't resolve the error by looking up the error code and the documentation, contact Salesforce for help and provide the rpcId value to Salesforce Customer Support.

Exception Example

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

```
[Trailer] = error-code [Value] =
sfdc.platform.eventbus.grpc.subscription.fetch.replayid.corrupted
```

```
=== GRPC Exception ===
```

```
io.grpc.StatusRuntimeException: INVALID_ARGUMENT: The Replay ID validation
failed. Ensure that the Replay ID is valid. rpcId:
2f6b4cee-3525-49d1-8fdb-0bd3d662062f
```

```
=== Trailers ===
```

```
[Trailer] = content-type [Value] = application/grpc
[Trailer] = rpc-id [Value] = 2f6b4cee-3525-49d1-8fdb-0bd3d662062f
[Trailer] = error-code [Value] =
sfdc.platform.eventbus.grpc.subscription.fetch.replayid.corrupted
[Trailer] = type [Value] = Subscribe
```

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 | |
| <code>sfdc.platform.eventbus.grpc.service.unavailable</code> | The Pub/Sub API service is unavailable. |

| | |
|-----------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sfdc.platform.eventbus.grpc.service.auth.error</code> | An authentication exception occurred. Provide valid authentication via metadata headers. |
| <code>sfdc.platform.eventbus.grpc.service.auth.headers.invalid</code> | The auth headers value for the specified key is invalid. Provide valid auth headers. The auth headers can't be blank. |
| <code>sfdc.platform.eventbus.grpc.service.auth.refresh.invalid</code> | The auth refresh token value is invalid. Provide a valid auth refresh value. |
| <code>sfdc.platform.eventbus.grpc.service.protection.triggered</code> | The service received too many connections and doesn't have the resources to accept new connections. |
| <code>sfdc.platform.eventbus.grpc.service.tenant.license</code> | <p>The Salesforce org is not licensed to access the Pub/Sub API. Contact Salesforce to enable the API.</p> <p>Note If you get this error even though your org is enabled for the Pub/Sub API pilot, retry the RPC method call. The cause of this error can be an intermittent issue with the service when it fails to verify the org's access to the pilot.</p> |
| Schema errors | |
| <code>sfdc.platform.eventbus.grpc.schema.meta.permission</code> | An error occurred while getting the schema. Possible reasons are an incorrect schema ID or incorrect credentials. |
| <code>sfdc.platform.eventbus.grpc.schema.api.unavailable</code> | The schema information is unavailable. |
| <code>sfdc.platform.eventbus.grpc.schema.validation.failed</code> | Schema validation failed. The schema ID can't be blank. |
| Topic errors | |
| <code>sfdc.platform.eventbus.grpc.topic.meta.permission</code> | An error occurred while getting the metadata for the specified topic. Ensure the credentials and topic name are correct. |
| <code>sfdc.platform.eventbus.grpc.topic.not.found</code> | The topic information for the specified topic was not found. Ensure the topic name is correct and that the topic exists. |
| <code>sfdc.platform.eventbus.grpc.topic</code> | The topic information is unavailable for the |

| | |
|----------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| <code>.api.unavailable</code> | specified topic. |
| <code>sfdc.platform.eventbus.grpc.topic.validation.empty</code> | An error occurred while validating the topic information provided. The topic can't be blank. |
| Publish errors | |
| <code>sfdc.platform.eventbus.grpc.publish.event.count.invalid</code> | The request contains no events. |
| <code>sfdc.platform.eventbus.grpc.publish.topic.mismatch</code> | There is a mismatch of topic names between requests. |
| <code>sfdc.platform.eventbus.grpc.publish.auth.refresh.invalid</code> | The refresh token shouldn't be provided in the initial request. |
| <code>sfdc.platform.eventbus.grpc.publish.topic.validation.empty</code> | An error occurred while validating the provided topic information. The topic can't be blank. |
| <code>sfdc.platform.eventbus.grpc.publish.stream.sweeper.timeout</code> | No publish request was received during the timeout period of 120 seconds. The publish stream timed out. |
| Subscription errors | |
| <code>sfdc.platform.eventbus.grpc.subscription.fetch.requested.events.invalid</code> | The requested number of events in a fetch request must be greater than zero. |
| <code>sfdc.platform.eventbus.grpc.subscription.fetch.topic.mismatch</code> | There is a mismatch of topic names between requests. |
| <code>sfdc.platform.eventbus.grpc.subscription.fetch.replayid.validation.failed</code> | The Replay ID validation failed. Provide a Replay ID in the custom preset. |
| <code>sfdc.platform.eventbus.grpc.subscription.fetch.replayid.corrupted</code> | The Replay ID validation failed. Ensure that the Replay ID is valid. |
| <code>sfdc.platform.eventbus.grpc.subscription.topic.cannot.subscribe</code> | Can't subscribe to the specified topic. Check that you have the required permissions. |

| | |
|-----------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sfdc.platform.eventbus.grpc.subscription.fetch.replay.repeated</code> | 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. |
| <code>sfdc.platform.eventbus.grpc.subscription.fetch.overflow</code> | Too many total events were requested for this subscription. Process some events first before sending a new fetch request. |
| <code>sfdc.platform.eventbus.grpc.subscription.internal.error</code> | 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

We recommend that the total size of a batch of events in a `PublishRequest` doesn't exceed 3 MB. The 3 MB recommendation is below the gRPC 4 MB limit and ensures optimal performance. If your `PublishRequest` exceeds 4 MB, the publish call fails with the following gRPC error, and the server closes the stream.

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

Each event message in a batch in a `PublishRequest` has a maximum size of 1 MB. If you publish a batch of events containing an event larger than 1 MB, the `PublishResult` of that event contains an error, and the event can't be published. Each event in the batch has its own `PublishResult`, so it can still be published if there are no errors.

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](#)