Contents

TriggerContext Class ................................................................. 60
Standard Platform Event Objects ............................................... 61
Use platform events to deliver secure and scalable custom notifications within Salesforce or from external sources. Define fields to customize your platform event. Your custom platform event determines the event data that the Lightning platform can produce or consume.

IN THIS SECTION:

Delivering Custom Notifications with Platform Events
Platform events are part of Salesforce’s enterprise messaging platform. The platform provides an event-driven messaging architecture to enable apps to communicate inside and outside of Salesforce. Before diving into platform events, take a look at what an event-based software system is.

Defining Your Custom Platform Event
Custom platform events are sObjects, similar to custom objects. Define a platform event in the same way you define a custom object.

Publishing Platform Events
After a platform event has been defined in your Salesforce org, publish event messages from a Salesforce app using processes, flows, or Apex or an external app using Salesforce APIs.

Subscribing to Platform Events
Receive platform events in processes, flows, Apex triggers, or CometD clients.

Testing Your Platform Event in Apex
Add Apex tests to test platform event subscribers. Before you can package or deploy Apex code, including triggers, to production, it must have tests and sufficient code coverage. Add Apex tests to provide code coverage for your triggers.

Platform Event Considerations
Learn about special behaviors related to defining, publishing, and subscribing to platform events. Learn how to test platform events. And get an overview of the various events that Salesforce offers.

Examples
Check out platform event apps—an end-to-end example using a process and a flow, a Java client, and sample apps that cover business scenarios.

Reference
The reference documentation for platform events covers limits, an API object, and Apex methods.

Delivering Custom Notifications with Platform Events
Platform events are part of Salesforce’s enterprise messaging platform. The platform provides an event-driven messaging architecture to enable apps to communicate inside and outside of Salesforce. Before diving into platform events, take a look at what an event-based software system is.
IN THIS SECTION:

Event-Driven Software Architecture
An event-driven (or message-driven) software architecture consists of event producers, event consumers, and channels. The architecture is suitable for large distributed systems because it decouples event producers from event consumers, thereby simplifying the communication model in connected systems.

Enterprise Messaging Platform Events
The Salesforce enterprise messaging platform offers the benefits of event-driven software architectures. Platform events are the event messages (or notifications) that your apps send and receive to take further action. Platform events simplify the process of communicating changes and responding to them without writing complex logic. Publishers and subscribers communicate with each other through events. One or more subscribers can listen to the same event and carry out actions.

Event-Driven Software Architecture
An event-driven (or message-driven) software architecture consists of event producers, event consumers, and channels. The architecture is suitable for large distributed systems because it decouples event producers from event consumers, thereby simplifying the communication model in connected systems.

Event
A change in state that is meaningful in a business process. For example, a placement of a purchase order is a meaningful event because the order fulfillment center requires notification to process the order. Or a change in a refrigerator’s temperature can indicate that it needs service.

Event message
A message that contains data about the event. Also known as an event notification.

Event producer
The publisher of an event message over a channel.

Channel
A conduit in which an event producer transmits a message. Event consumers subscribe to the channel to receive messages. Also referred to as event bus in Salesforce.

Event consumer
A subscriber to a channel that receives messages from the channel.

Systems in request-response communication models make a request to a web service or database to obtain information about a certain state. The sender of the request establishes a connection to the service and depends on the availability of the service.

In comparison, systems in an event-based model obtain information and can react to it in near real time when the event occurs. Event producers don’t know the consumers that receive the events. Any number of consumers can receive and react to the same events. The only dependency between producers and consumers is the semantic of the message content.

The following diagram illustrates an event-based software architecture system.
Enterprise Messaging Platform Events

The Salesforce enterprise messaging platform offers the benefits of event-driven software architectures. Platform events are the event messages (or notifications) that your apps send and receive to take further action. Platform events simplify the process of communicating changes and responding to them without writing complex logic. Publishers and subscribers communicate with each other through events. One or more subscribers can listen to the same event and carry out actions.

For example, a software system can send events containing information about printer ink cartridges. Subscribers can subscribe to the events to monitor printer ink levels and place orders to replace cartridges with low ink levels.

Custom Platform Events

Use custom platform events to publish and process custom notifications. For example, publish custom platform events to send order information to an order fulfillment service. Or publish custom platform events to send printer ink information that is processed by a service app.

You define a custom platform event in Salesforce in the same way that you define a custom object. Create a platform event definition by giving it a name and adding custom fields. Platform events support a subset of field types in Salesforce. See Platform Event Fields. This table lists a sample definition of custom fields for a printer ink event.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field API Name</th>
<th>Field Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printer Model</td>
<td>Printer_Model__c</td>
<td>Text</td>
</tr>
<tr>
<td>Serial Number</td>
<td>Serial_Number__c</td>
<td>Text</td>
</tr>
<tr>
<td>Ink Percentage</td>
<td>Ink_Percentage__c</td>
<td>Number</td>
</tr>
</tbody>
</table>
You can publish custom platform events on the Lightning Platform by using Apex or point-and-click tools, such as Process Builder and Flow Builder, or an API in external apps. Similarly, you can subscribe to an event channel either on the platform through an Apex trigger or point-and-click tools, or in external apps using the CometD-based Streaming API. When an app publishes an event message, event subscribers receive the event message and execute business logic. Using the printer ink example, a software system monitoring a printer makes an API call to publish an event when the ink is low. The printer event message contains the printer model, serial number, and ink level. After the printer sends the event message, an Apex trigger is fired in Salesforce. The trigger creates a case record to place an order for a new cartridge.

**Standard Platform Events**

Salesforce provides events with predefined fields, called standard platform events. An example of a standard platform event is AssetTokenEvent, which monitors OAuth 2.0 authentication activity. Another example is BatchApexErrorEvent, which reports errors encountered in batch Apex jobs.

Salesforce publishes standard platform events in response to an action that occurred in the app or errors in batch Apex jobs. You can subscribe to a standard platform event stream using the subscription mechanism that the event supports.

**Platform Events and sObjects**

A platform event is a special kind of Salesforce entity, similar in many ways to an sObject. An event message is an instance of a platform event, similar to how a record is an instance of a custom or standard object. Unlike custom or standard objects, you can’t update or delete event records. You also can’t view event records in the Salesforce user interface, and platform events don’t have page layouts. When you delete a platform event definition, it’s permanently deleted.

You can set read and create permissions for platform events. Grant permissions to users in profiles or in permission sets.

**Platform Events and Transactions**

Unlike custom objects, platform events aren’t processed within database transactions in the Lightning Platform. As a result, published platform events can’t be rolled back. Note the following:

- The allOrNoneHeader API header is ignored when publishing platform events through the API.
- The Apex setSavepoint() and rollback()Database methods aren’t supported with platform events.
- The publishing of high-volume platform events is asynchronous. For more information, see Asynchronous Publishing.

When publishing platform events, DML limits and other Apex governor limits apply.

**Event Retention**

High-volume platform event messages are stored for 72 hours (3 days). Standard-volume platform event messages are stored for 24 hours (1 day). You can retrieve past event messages when using CometD clients to subscribe to a channel.

**High-Volume Platform Events**

Use high-volume platform events to publish and process millions of events efficiently and to scale your event-based apps. Previously, standard-volume events were available. In API version 45.0 and later, your new custom event definitions are high volume by default. Standard-volume events are still supported but not available for new event definitions. High-volume platform events offer better scalability than standard-volume platform events.

Note the following characteristics of high-volume platform events.
Asynchronous Publishing

For efficient processing of high loads of incoming event messages, high-volume platform events are published asynchronously. After the publishing call returns with a successful result, the publish request is queued in Salesforce. The event message might not be published immediately. For more information, see High-Volume Platform Event Persistence.

Separate Event Allocations

Each Salesforce edition provides default allocations and usage-based entitlements for the number of high-volume events delivered monthly to CometD clients. See Platform Event Allocations.

SEE ALSO:

Standard Platform Event Objects

Defining Your Custom Platform Event

Custom platform events are sObjects, similar to custom objects. Define a platform event in the same way you define a custom object.

IN THIS SECTION:

Platform Event Fields
- Platform events contain standard fields. Add custom fields for your custom data.
- Migrate Platform Event Definitions with Metadata API
- Deploy and retrieve platform event definitions from your sandbox and production org as part of your app’s development life cycle.

Platform Event Fields

Platform events contain standard fields. Add custom fields for your custom data.

To define a platform event in Salesforce Classic or Lightning Experience:

1. From Setup, enter Platform Events in the Quick Find box, then select Platform Events.
3. Complete the standard fields, and optionally add a description.
4. For Event Type, select High Volume.
5. Click Save.
6. To add a field, in the Custom Fields & Relationships related list, click New.
7. Follow the custom field wizard to set up the field properties.

Note: In Lightning Experience, platform events aren’t shown in the Object Manager’s list of standard and custom objects and aren’t available in Schema Builder.

Standard Fields

Platform events include standard fields. These fields appear on the New Platform Event page.
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>Name used to refer to your platform event in a user interface page.</td>
</tr>
<tr>
<td>Plural Label</td>
<td>Plural name of the platform event.</td>
</tr>
<tr>
<td>Starts with a vowel sound</td>
<td>If it’s appropriate for your org’s default language, indicate whether the label is preceded by “an” instead of “a.”</td>
</tr>
<tr>
<td>Object Name</td>
<td>Unique name used to refer to the platform event when using the API. In managed packages, this name prevents naming conflicts with package installations. Use only alphanumeric characters and underscores. The name must begin with a letter and have no spaces. It cannot end with an underscore nor have two consecutive underscores.</td>
</tr>
<tr>
<td>Description</td>
<td>Optional description of the object. A meaningful description helps you remember the differences between your events when you are viewing them in a list.</td>
</tr>
<tr>
<td>Deployment Status</td>
<td>Indicates whether the platform event is visible to other users.</td>
</tr>
</tbody>
</table>

### Custom Fields

In addition to the standard fields, add custom fields to customize your event. Platform event custom fields support only these field types:

- Checkbox
- Date
- Date/Time
- Number
- Text
- Text Area (Long)

The maximum number of fields that you can add to a platform event is the same as for a custom object. See Salesforce Features and Edition Allocations.

### ReplayId System Field

Each event message is assigned an opaque ID contained in the `ReplayId` field. The `ReplayId` field value, which is populated by the system when the event is delivered to subscribers, refers to the position of the event in the event stream. Replay ID values are not guaranteed to be contiguous for consecutive events. For example, the event following the event with ID 999 can have an ID of 1,025. A subscriber can store a replay ID value and use it on resubscription to retrieve events that are within the retention window. For example, a subscriber can retrieve missed events after a connection failure. Subscribers must not compute new replay IDs based on a stored replay ID to refer to other events in the stream.
API Name Suffix for Platform Events

When you create a platform event, the system appends the __e suffix to create the API name of the event. For example, if you create an event with the object name Low__Ink, the API name is Low_Ink__e. The API name is used whenever you refer to the event programmatically, for example, in Apex.

SEE ALSO:
- Considerations for Defining and Publishing Platform Events
- Considerations for Publishing and Subscribing to Platform Events with Apex and APIs

Migrate Platform Event Definitions with Metadata API

Deploy and retrieve platform event definitions from your sandbox and production org as part of your app's development life cycle. The CustomObject metadata type represents a platform event. Platform event names are appended with __e. The file that contains the platform event definition has the suffix .object. Platform events are stored in the objects folder.

Example: Here is a definition of a platform event with a number field and two text fields.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<CustomObject xmlns="http://soap.sforce.com/2006/04/metadata">
  <deploymentStatus>Deployed</deploymentStatus>
  <eventType>HighVolume</eventType>
  <fields>
    <fullName>Ink_Percentage__c</fullName>
    <externalId>false</externalId>
    <isFilteringDisabled>false</isFilteringDisabled>
    <isNameField>false</isNameField>
    <isSortingDisabled>false</isSortingDisabled>
    <label>Ink Percentage</label>
    <precision>18</precision>
    <required>false</required>
    <scale>2</scale>
    <type>Number</type>
    <unique>false</unique>
  </fields>
  <fields>
    <fullName>Printer_Model__c</fullName>
    <externalId>false</externalId>
    <isFilteringDisabled>false</isFilteringDisabled>
    <isNameField>false</isNameField>
    <isSortingDisabled>false</isSortingDisabled>
    <label>Printer Model</label>
    <length>20</length>
    <required>false</required>
    <type>Text</type>
    <unique>false</unique>
  </fields>
  <fields>
    <fullName>Serial_Number__c</fullName>
    <externalId>false</externalId>
    <isFilteringDisabled>false</isFilteringDisabled>
```

Migrate Platform Event Definitions with Metadata API
The `eventType` field specifies the platform event volume. Only the `HighVolume` value is supported. The `StandardVolume` value is deprecated. If you create a platform event with the `StandardVolume` event type, you get an error.

This package.xml manifest file references the previous event definition. The name of the referenced event is `Low_Ink__e`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
  <types>
    <members>Low_Ink__e</members>
    <name>CustomObject</name>
  </types>
  <version>45.0</version>
</Package>
```

Retrieving Platform Events

To retrieve all platform events, in addition to custom objects defined in your org, use the wildcard character (*) for the `<members>` element, as follows.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
  <types>
    <members>*</members>
    <name>CustomObject</name>
  </types>
  <version>45.0</version>
</Package>
```

To retrieve or deploy triggers associated to a platform event, use the ApexTrigger metadata type. For more information about how to use Metadata API and its types, see the Metadata API Developer Guide.

**Publishing Platform Events**

After a platform event has been defined in your Salesforce org, publish event messages from a Salesforce app using processes, flows, or Apex or an external app using Salesforce APIs.

**IN THIS SECTION:**

- Publish Event Messages with Processes
  Use Process Builder to publish event messages from a Salesforce app as part of an automated process.
Publish Event Messages with Flows
Use flows to publish event messages from a Salesforce app as part of some user interaction, an automated process, Apex, or workflow action.

Publish Event Messages with Apex
Use Apex to publish event messages from a Salesforce app.

Publish Event Messages with Salesforce APIs
External apps use an API to publish platform event messages.

SEE ALSO:
Decoupled Publishing and Subscription

Publish Event Messages with Processes
Use Process Builder to publish event messages from a Salesforce app as part of an automated process.

To publish event messages, add a Create a Record action to the appropriate process. Where you’d usually pick an object to create, select the custom platform event.

For example, here’s how to configure a Create a Record action that publishes a Low Ink event message. This example assumes that the Low Ink platform event is defined in your org and that the event has these custom fields.

- Printer Model (Text)
- Serial Number (Text)
- Ink Percentage (Number)

1. For Record Type, enter Low and select Low Ink.
2. Set the field values.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printer Model</td>
<td>String</td>
<td>XZO-5</td>
</tr>
<tr>
<td>Serial Number</td>
<td>String</td>
<td>12345</td>
</tr>
<tr>
<td>Ink Percentage</td>
<td>Number</td>
<td>0.2</td>
</tr>
</tbody>
</table>
3. Save the action and activate the process.

SEE ALSO:

Salesforce Help: Lightning Process Builder

Publish Event Messages with Flows

Use flows to publish event messages from a Salesforce app as part of some user interaction, an automated process, Apex, or workflow action.

To publish event messages, add a Create Records element to the appropriate flow. Where you’d usually pick an object to create, select the custom platform event.

For example, here’s how to configure a Create Records element that publishes a Printer Status platform event message. This example assumes that the Printer Status platform event is defined in your org and that the event has these custom fields.

- Printer Model (Text)
- Serial Number (Text)
- Ink Status (Text)

1. For How Many Records to Create, choose One.
2. For How to Set the Record Fields, choose Use separate variables, resources, and literal values.
3. For Object, enter Printer and select Printer Status.
4. Set these field values.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printer Model</td>
<td>XZO-5</td>
</tr>
<tr>
<td>Serial Number</td>
<td>12345</td>
</tr>
<tr>
<td>Ink Status</td>
<td>Low</td>
</tr>
</tbody>
</table>
5. Save and activate the flow.

SEE ALSO:
Salesforce Help: Flows

Publish Event Messages with Apex

Use Apex to publish event messages from a Salesforce app.

To publish event messages, call the EventBus.publish method. For example, if you’ve defined a custom platform event called Low_Ink, reference this event type as Low_Ink__e. Next create instances of this event and pass them to the Apex method.

Example: This example creates two events of type Low_Ink__e, publishes them, and then checks whether the publishing was successful or errors were encountered.

Before you can run this snippet, define a platform event with the name of Low_Ink__e and the following fields: Printer_Model__c of type Text, Serial_Number__c of type Text (marked as required), Ink_Percentage__c of type Number(16, 2).

```java
List<Low_Ink__e> inkEvents = new List<Low_Ink__e>(){
    inkEvents.add(new Low_Ink__e(Printer_Model__c='XZO-5', Serial_Number__c='12345', Ink_Percentage__c=0.2));
    inkEvents.add(new Low_Ink__e(Printer_Model__c='MN-123', Serial_Number__c='10013', Ink_Percentage__c=0.15));
};
```
// Call method to publish events
List<Database.SaveResult> results = EventBus.publish(inkEvents);

// Inspect publishing result for each event
for (Database.SaveResult sr : results) {
    if (sr.isSuccess()) {
        System.debug('Successfully published event.');
    } else {
        for (Database.Error err : sr.getErrors()) {
            System.debug('Error returned: ' + err.getStatusCode() + ' - ' + err.getMessage());
        }
    }
}

For each event, Database.SaveResult contains information about whether the operation was successful and the errors encountered. If the isSuccess() method returns true, the event was published for a standard-volume event. For a high-volume event, the publish request is queued in Salesforce and the event message might not be published immediately. For more information, see High-Volume Platform Event Persistence. If isSuccess() returns false, the event publish operation resulted in errors which are returned in the Database.Error object. EventBus.publish() can publish some passed-in events, even when other events can't be published due to errors. The EventBus.publish() method doesn't throw exceptions caused by an unsuccessful publish operation. It is similar in behavior to the Apex Database.insert method when called with the partial success option.

Database.SaveResult also contains the Id system field. The Id field value is not included in the event message delivered to subscribers. It is not used to identify an event message, and is not always unique.

The event insertion occurs non-transactionally. As a result, you can't roll back published events. Because event publishing is equivalent to a DML insert operation, DML limits and other Apex governor limits apply.

SEE ALSO:
- EventBus Class
- Platform Event Error Status Codes

Publish Event Messages with Salesforce APIs

External apps use an API to publish platform event messages.

Publish events by creating records of your event in the same way that you insert sObjects. You can use any Salesforce API to create platform events, such as SOAP API, REST API, or Bulk API.

When publishing an event message, the result that the API returns contains information about whether the operation was successful and the errors encountered. If the success field is true, the event was published for a standard-volume event. For a high-volume event, the publish request is queued in Salesforce and the event message might not be published immediately. For more details, see High-Volume Platform Event Persistence. If the success field is false, the event publish operation resulted in errors, which are returned in the errors field.

The returned result also contains the Id system field. The Id field value is not included in the event message delivered to subscribers. It is not used to identify an event message, and is not always unique. Subscribers can use the ReplayId system field, which is included in the delivered message, to identify the position of the event in the stream.
REST API

To publish a platform event message using REST API, send a POST request to the following endpoint.

/services/data/v45.0/sobjects/Event_Name__e/

**Example:** If you’ve defined a platform event named Low Ink, publish event notifications by inserting Low_Ink__e records.

REST endpoint:
/services/data/v45.0/sobjects/Low_Ink__e/

Request body:

```javascript
{
    "Printer_Model__c" : "XZO-5"
}
```

After the platform event record is created, the REST response looks like this output. Headers are deleted for brevity.

```
HTTP/1.1 201 Created
{
    "id" : "e00xx000000000B",
    "success" : true,
    "errors" : [ ]
}
```

SOAP API

To publish a platform event message using SOAP API, use the `create()` call.

**Example:** This example shows the SOAP message (using Partner API) of a request to create three platform events in one call. Each event has one custom field named Printer_Model__c.

```xml
<?xml version="1.0" encoding="UTF-8"?>
xmlns:ns1="urn:sobject.partner.soap.sforce.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ns2="urn:partner.soap.sforce.com">
  <SOAP-ENV:Header>
    <ns2:SessionHeader>
      <ns2:sessionId>00DR00000001fWV!AQMAQOshATCQ4fBaYFOTrHVixfEO61...</ns2:sessionId>
    </ns2:SessionHeader>
    <ns2:CallOptions>
      <ns2:client>Workbench/34.0.12i</ns2:client>
      <ns2:defaultNamespace xsi:nil="true"/>
      <ns2:returnFieldDataTypes xsi:nil="true"/>
    </ns2:CallOptions>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns2:create>
      <ns2:sObjects>
        <ns1:type>Low_Ink__e</ns1:type>
```

The response of the Partner SOAP API request looks something like the following. Headers are deleted for brevity.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns="urn:partner.soap.sforce.com">
  <soapenv:Header>
    ...
  </soapenv:Header>
  <soapenv:Body>
    <createResponse>
      <result>
        <id>e00xx000000000F</id>
        <success>true</success>
      </result>
      <result>
        <id>e00xx000000000G</id>
        <success>true</success>
      </result>
      <result>
        <id>e00xx000000000H</id>
        <success>true</success>
      </result>
    </createResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

SEE ALSO:
- REST API Developer Guide
- SOAP API Developer Guide: create() call
- Bulk API Developer Guide
- Platform Event Error Status Codes
Subscribing to Platform Events

Receive platform events in processes, flows, Apex triggers, or CometD clients.

IN THIS SECTION:

Set Up Debug Logs for Event Subscriptions

Debug logs for platform event triggers, event processes, and resumed flow interviews are created by “Automated Process” and are separate from their corresponding Apex code logs. The debug logs aren’t available in the Developer Console’s Log tab. To collect logs for an event subscription, add a trace flag entry for the Automated Process entity in Setup.

Subscribe to Platform Event Messages with Processes

Processes can subscribe to platform events and receive event messages published through Apex, APIs, flows, or other processes. Processes provide an autosubscription mechanism.

Subscribe to Platform Event Messages with Flows

Running instances of flows, called interviews, can subscribe to platform events and receive event messages published through Apex, APIs, flows, or other processes. Flows provide an autosubscription mechanism.

Subscribe to Platform Event Notifications with Apex Triggers

Use Apex triggers to subscribe to events. You can receive event notifications in triggers regardless of how they were published—through Apex or APIs. Triggers provide an autosubscription mechanism. No need to explicitly create and listen to a channel in Apex.

Subscribe to Platform Event Notifications in an Aura Component

Add the lightning:empApi component to your custom Aura component to receive platform event messages. The lightning:empApi component provides access to methods for subscribing to a streaming channel and listening to event messages.

Subscribe to Platform Event Notifications with CometD

Use CometD to subscribe to platform events in an external client. Implement your own CometD client or use EMP Connector, an open-source, community-supported tool that implements all the details of connecting to CometD and listening on a channel.

Obtain a Platform Event’s Subscribers

View a list of all triggers or processes that are subscribed to a platform event by using the Salesforce user interface or the API.

SEE ALSO:

Decoupled Publishing and Subscription

Set Up Debug Logs for Event Subscriptions

Debug logs for platform event triggers, event processes, and resumed flow interviews are created by “Automated Process” and are separate from their corresponding Apex code logs. The debug logs aren’t available in the Developer Console’s Log tab. To collect logs for an event subscription, add a trace flag entry for the Automated Process entity in Setup.

1. From Setup, enter Debug Logs in the Quick Find box, then click Debug Logs.
2. Click New.
3. For Traced Entity Type, select Automated Process.
4. Select the time period to collect logs and the debug level.
5. Click Save.
To collect logs for the user who publishes the events, add another trace flag entry for that user.

SEE ALSO:

*Salesforce Help: Set Up Debug Logging*

**Subscribe to Platform Event Messages with Processes**

Processes can subscribe to platform events and receive event messages published through Apex, APIs, flows, or other processes. Processes provide an autosubscription mechanism.

To subscribe a process to a platform event, build the process to start when it receives a platform event message. In the process’s trigger, associate the process with a platform event and an object.

**Example:** This process starts when it receives a Printer Status event message. When it starts, the process looks for an Asset record whose serial number matches the serial number in the event message.

If flow interviews and active processes are subscribed to the same platform event, we can’t guarantee which one processes the event message first.

A process evaluates platform event messages in the order they’re received. The order of event messages is based on the event replay ID. A process can receive a batch of event messages at once. The order is preserved within each batch. The event messages in a batch can originate from multiple publishers.

Unlike record-change processes, event processes don’t execute in the same Apex transaction as whatever published the event message. The process runs asynchronously under the Automated Process entity. As a result, there can be a delay between when an event message is published and when the process evaluates the event message. Automated Process creates the debug logs corresponding to the process execution, but the actions are performed on behalf of the user who published the event. System fields, such as `CreatedById` and `LastModifiedById`, reference the user who published the event message.
Event processes and record-change processes have similar limitations. For example, they’re both subject to Apex governor limits.

SEE ALSO:
- Considerations for Subscribing to Platform Events with Processes and Flows
- Salesforce Help: Process Limits and Considerations
- Set Up Debug Logs for Event Subscriptions
- Obtain Processes That Subscribe to a Platform Event

Subscribe to Platform Event Messages with Flows

Running instances of flows, called interviews, can subscribe to platform events and receive event messages published through Apex, APIs, flows, or other processes. Flows provide an autosubscription mechanism.

To build a flow that subscribes to a platform event at run time, add a Pause element and set it up as follows.

- (Optional) Specify conditions that determine whether to pause a flow interview.
- Select the platform event that the flow interview subscribes to.
- Identify the values that a received event message must have to resume the flow interview.
- (Optional) Create a record variable in the flow to store the data from the event message that resumes the flow interview.

Example: This pause element is set up to resume a flow interview when a vendor response event message is received (1). The order number in the event message must match the flow’s {!orderNumber} variable value, and the order status must be Shipped (2). When the flow interview resumes, the {!vendorResponse} record variable is populated with the data from the event message (3).
If flow interviews and active processes are subscribed to the same platform event, we can’t guarantee which one processes the event message first.

Flow interviews evaluate platform event messages in the order they’re received. The order of event messages is based on the event replay ID. A flow interview can receive a batch of event messages at once. The order is preserved within each batch. The event messages in a batch can originate from multiple publishers.

Flow interviews resume in a separate Apex transaction than the transaction that published the event message. The flow interview resumes asynchronously under the Automated Process entity. As a result, there can be a delay between when an event message is published and when the interview evaluates the event message. Automated Process creates the debug logs corresponding to the interview resuming, but the interview’s actions are executed on behalf of the user who published the event message. System fields, such as CreatedById and LastModifiedById, reference the user who published the event message.

SEE ALSO:
- Considerations for Subscribing to Platform Events with Processes and Flows
- Salesforce Help: Flow Limits and Considerations
- Salesforce Help: Paused Flow Interview Considerations
Subscribe to Platform Event Notifications with Apex Triggers

Use Apex triggers to subscribe to events. You can receive event notifications in triggers regardless of how they were published—through Apex or APIs. Triggers provide an autosubscription mechanism. No need to explicitly create and listen to a channel in Apex.

To subscribe to event notifications, write an after insert trigger on the event object type. The after insert trigger event corresponds to the time after a platform event is published. After an event message is published, the after insert trigger is fired.

**Example:** This example shows a trigger for the Low Ink event. It iterates through each event and checks the Printer_Model__c field value. The trigger inspects each received notification and gets the printer model from the notification. If the printer model matches a certain value, other business logic is executed. For example, the trigger creates a case to order a new cartridge for this printer model.

```apex
// Trigger for catching Low_Ink events.
trigger LowInkTrigger on Low_Ink__e (after insert) {
    // List to hold all cases to be created.
    List<Case> cases = new List<Case>();

    // Get user Id for case owner. Replace username value with a valid value.
    User adminUser = [SELECT Id FROM User WHERE Username='admin@acme.org'];

    // Iterate through each notification.
    for (Low_Ink__e event : Trigger.New) {
        System.debug('Printer model: ' + event.Printer_Model__c);
        if (event.Printer_Model__c == 'MN-123') {
            // Create Case to order new printer cartridge.
            Case cs = new Case();
            cs.Priority = 'Medium';
            cs.Subject = 'Order new ink cartridge for SN ' + event.Serial_Number__c;
            // Set case owner ID so it is not set to the Automated Process entity.
            cs.OwnerId = adminUser.Id;
            cases.add(cs);
        }
    }

    // Insert all cases in the list.
    if (cases.size() > 0) {
        insert cases;
    }
}
```

An Apex trigger processes platform event notifications sequentially in the order they’re received. The order of events is based on the event replay ID. An Apex trigger can receive a batch of events at once. The order of events is preserved within each batch. The events in a batch can originate from one or more publishers.

Unlike triggers on standard or custom objects, triggers on platform events don’t execute in the same Apex transaction as the one that published the event. The trigger runs asynchronously in its own process under the Automated Process entity. As a result, there might be a delay between when an event is published and when the trigger processes the event. Also, debug logs corresponding to the trigger execution are created by Automated Process. System fields, such as CreatedById and LastModifiedById, reference the Automated Process entity.

**Note:** If you create a Salesforce record with an ownerId field in the trigger, such as a case or opportunity, explicitly set the owner ID. For cases and leads, you can alternatively use assignment rules to set the owner. See Considerations for Publishing and Subscribing to Platform Events with Apex and APIs.
Event triggers have many of the same limitations of custom and standard object triggers. For example, with some exceptions, you generally can’t make Apex callouts from triggers. For more information, see Implementation Considerations for triggers in the Apex Developer Guide.

Platform Event Triggers and Apex Governor Limits

Platform event triggers are subject to Apex governor limits.

**Synchronous Governor Limits**

When governor limits are different for synchronous and asynchronous Apex, the synchronous limits apply to platform event triggers. Asynchronous limits are for long-lived processes, such as Batch Apex and future methods. Synchronous limits are for short-lived processes that execute quickly. Although platform event triggers run asynchronously, they’re short-lived processes that execute in batches rather quickly.

**Reset Limits**

Because a platform event trigger runs in a separate transaction from the one that fired it, governor limits are reset, and the trigger gets its own set of limits.

IN THIS SECTION:

- **Retry Event Triggers with EventBus.RetryableException**
  Get another chance to process event notifications. Retrying a trigger is helpful when a transient error occurs or when waiting for a condition to change. Retry a trigger if the error or condition is external to the event records and is likely to go away later.

- **Email Notifications for Triggers in Error State**
  When an Apex platform event trigger exceeds the maximum number of retries and is in the error state, you are emailed a notification. To resume processing of new event messages, fix and save the trigger. Event messages sent while the trigger is stopped are missed and can’t be processed again.

SEE ALSO:

- Apex Developer Guide: Execution Governors and Limits
- Set Up Debug Logs for Event Subscriptions
- Obtain Apex Triggers That Subscribe to a Platform Event
- Considerations for Publishing and Subscribing to Platform Events with Apex and APIs

**Retry Event Triggers with EventBus.RetryableException**

Get another chance to process event notifications. Retrying a trigger is helpful when a transient error occurs or when waiting for a condition to change. Retry a trigger if the error or condition is external to the event records and is likely to go away later.

An example of a transient condition: A trigger adds a related record to a master record if a field on the master record equals a certain value. It is possible that in a subsequent try, the field value changes and the trigger can perform the operation.

To retry the event trigger, throw EventBus.RetryableException. Events are resent after a small delay. The delay increases in subsequent retries. If the trigger receives a batch of events, retrying the trigger causes all events in the batch to be resent. Resent events have the same field values as the original events, but the batch sizes of the events can differ. For example, the initial trigger can receive events with replay ID 10 to 20. The resent batch can be larger, containing events with replay ID 10 to 40. When the trigger is retried, the DML operations performed in the trigger before the retry are rolled back and no changes are saved.
Limit the Number of Retry Attempts

You can run a trigger up to 10 times when it is retried (the initial run plus nine retries). After the trigger is retried nine times, it moves to the error state and stops processing new events. Events sent after the trigger moves to the error state and before it returns to the running state are not resent to the trigger. To resume event processing, fix the trigger and save it.

We recommend limiting the retries to less than nine times. Use the EventBus.TriggerContext.currentContext().retries property to check how many times the trigger has been retried. Alternatively, you can query the EventBusSubscriber.retries field in API version 43.0 and later.

Example: This example is a skeletal trigger that gives you an idea of how to throw EventBus.RetryableException and limit the number of retries. The trigger uses an if statement to check whether a certain condition is true. Alternatively, you can use a try-catch block and throw EventBus.RetryableException in the catch block.

```java
trigger ResendEventsTrigger on Low_Ink__e (after insert) {
    if (condition == true) {
        // Process platform events.
    } else {
        // Ensure we don't retry the trigger more than 4 times
        if (EventBus.TriggerContext.currentContext().retries < 4) {
            // Condition isn't met, so try again later.
            throw new EventBus.RetryableException('Condition is not met, so retrying the trigger again.);
        } else {
            // Trigger was retried enough times so give up and
            // resort to alternative action.
            // For example, send email to user.
        }
    }
}
```

Email Notifications for Triggers in Error State

When an Apex platform event trigger exceeds the maximum number of retries and is in the error state, you are emailed a notification. To resume processing of new event messages, fix and save the trigger. Event messages sent while the trigger is stopped are missed and can't be processed again.

The email notification is not sent for general unhandled exceptions, such as uncatchable limit exceptions. Unlike Apex object triggers, platform event triggers don't generate exception emails for unhandled exceptions.

For a platform event trigger in the error state, the notification is sent to the developer specified in the trigger’s Last Modified By field. To also send the email to other users, add them on the Apex Exception Email page in Setup. The recipients specified on the Apex Exception Email page also apply to emails sent for Apex object triggers and classes.

To set up more recipients:

1. From Setup, enter Apex Exception Email in the Quick Find box.
2. Select Apex Exception Email.

The users and email addresses entered apply to all managed packages in the customer's org. You can also configure Apex exception emails using the Tooling API object ApexEmailNotification.
Subscribe to Platform Event Notifications in an Aura Component

Add the `lightning:empApi` component to your custom Aura component to receive platform event messages. The `lightning:empApi` component provides access to methods for subscribing to a streaming channel and listening to event messages.

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

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

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

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

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

Note: As of Spring '19 (API version 45.0), you can build Lightning components using two programming models: the Lightning Web Components model, and the original Aura Components model. Lightning web components are custom HTML elements built using HTML and modern JavaScript. Lightning web components and Aura components can coexist and interoperate on a page. This developer guide covers Aura components only.

Subscribe to Platform Event Notifications with CometD

Use CometD to subscribe to platform events in an external client. Implement your own CometD client or use EMP Connector, an open-source, community-supported tool that implements all the details of connecting to CometD and listening on a channel.

Salesforce sends platform events to CometD clients sequentially in the order they're received. The order of event notifications is based on the replay ID of events.

The process of subscribing to platform event notifications through CometD is similar to subscribing to PushTopics or generic events. The only difference is the channel name. The platform event channel name is case-sensitive and is in the following format.

```
/event/Event_Name__e
```

Use this CometD endpoint with the API version appended to it.

```
/cometd/45.0
```

Example: If you have a platform event named Low Ink, provide this channel name when subscribing.

```
/event/Low_Ink__e
```

The message of a delivered platform event looks similar to the following example for Low Ink events.

```json
{
  "data": {
    "schema": "dffQ2QLzDNHqwB8_sHMxdA",
    "payload": {
      "CreatedAt": "2017-04-09T18:31:40.517Z",
      "CreatedBy": "005D0000001cSZs",
      "Printer_Model__c": "XZO-5",
      "Serial_Number__c": "12345",
      "Ink_Percentage__c": 0.2
    }
  },
  "event": {
    "replayId": 2
  }
}
```
The schema field in the event message contains the ID of the platform event schema. The schema is versioned—when the schema changes, the schema ID changes as well.

To determine if the schema of an event has changed, retrieve the schema through REST API. Use the schema ID by performing a GET request to this REST API resource: `/vXX.X/event/eventSchema/{Schema_ID}`. Alternatively, you can retrieve the event schema by supplying the event name to this endpoint: `/vXX.X/sobjects/{Event_Name}/eventSchema`. For more information, see:

- **Platform Event Schema by Schema ID** in the *REST API Developer Guide*
- **Platform Event Schema by Event Name** in the *REST API Developer Guide*

You can use EMP Connector to receive delivered events. The connector subscribes to any type of streaming event and accepts the event channel name as an argument. See Example: Subscribe to and Replay Events Using a Java Client.

Add custom logic to your client to perform some operations after a platform event notification is received. For example, the client can create a request to order a new cartridge for this printer model.

**SEE ALSO:**
- CometD
  - Considerations for Publishing and Subscribing to Platform Events with Apex and APIs

### Obtain a Platform Event’s Subscribers

View a list of all triggers or processes that are subscribed to a platform event by using the Salesforce user interface or the API.

**Note:** CometD subscribers to a platform event channel aren’t exposed in the user interface or the API. Flow Pause element subscribers to a platform event aren’t returned in Metadata API.

**IN THIS SECTION:**

- **Obtain Processes That Subscribe to a Platform Event**
  Event processes aren’t reflected individually in the Subscriptions related list on the platform event definition. To get a list of processes that are subscribed to a platform event, use the Metadata API.

- **Obtain Apex Triggers That Subscribe to a Platform Event**
  View a list of all triggers that are subscribed to a platform event by using the Salesforce user interface or the API.

### Obtain Processes That Subscribe to a Platform Event

Event processes aren’t reflected individually in the Subscriptions related list on the platform event definition. To get a list of processes that are subscribed to a platform event, use the Metadata API.

1. Retrieve all event subscriptions in your org with this sample package manifest.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
  <types>
    <members>*</members>
    <name>EventSubscription</name>
  </types>
</Package>
```
2. In each .subscription file, look at the referenceData parameter. The value is the API name of a process.

Example: In this .subscription file, referenceData points to version 4 of the Printer_Management process.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<EventSubscription xmlns="http://soap.sforce.com/2006/04/metadata">
  <active>true</active>
  <eventType>Printer_Status__e</eventType>
  <referenceData>Printer_Management_4</referenceData>
</EventSubscription>
```

Obtain Apex Triggers That Subscribe to a Platform Event

View a list of all triggers that are subscribed to a platform event by using the Salesforce user interface or the API.

1. From Setup, enter Platform Events in the Quick Find box, then select Platform Events.

2. Click your event’s name.

On the event’s definition page, the Subscriptions related list shows all the active triggers that are subscribed to platform events. The list shows the replay ID of the event that the system last processed (Last Processed Id field) and the event last published (Last Published Id field). Knowing which replay ID was last processed is useful when there is a gap in the events published and processed. For example, if a trigger contains complex logic that causes a delay in processing large batches of incoming events.

Note: For high-volume platform events, the Last Published Id value is not available and is always -1.

Also, the Subscriptions list shows the state of each subscribed trigger. Trigger states can be one of the following.

- **Running**—The trigger is actively listening to events.
- **Idle**—The trigger hasn’t received events for some time and is not actively listening to events. When new events are sent, the trigger receives the new events after a short delay and switches to the Running state.
- **Error**—The trigger has been disconnected and stopped receiving published events. A trigger reaches this state when it exceeds the number of maximum retries with the EventBus.RetryableException. Trigger assertion failures and unhandled exceptions don’t cause the Error state. To resume trigger execution, fix the trigger code and save it. For a managed package trigger, redeploy the package after fixing the trigger.
- **Suspended**—The trigger is disconnected and can’t receive events due to an internal error. To resume trigger execution, try saving the trigger again. For a managed package trigger, redeploy the package.

Note: CometD subscribers to a platform event channel aren’t listed in the Subscriptions related list. Only one “Process” subscriber appears in the Subscriptions related list for all processes and flows that are subscribed to the platform event.

Alternatively, you can obtain the same subscriber information by querying the EventBusSubscriber object. See EventBusSubscriber.

Testing Your Platform Event in Apex

Add Apex tests to test platform event subscribers. Before you can package or deploy Apex code, including triggers, to production, it must have tests and sufficient code coverage. Add Apex tests to provide code coverage for your triggers.
Event and Event Bus Properties in Test Context

In test context, event messages and the event bus have different properties. State information of events and subscribers is reset and is not persisted.

Deliver Test Event Messages

Deliver test event messages after the `Test.stopTest()` statement. Alternatively, deliver test event messages at any time with the `Test.getEventBus().deliver()` method.

Test Retried Event Messages

An Apex trigger can retry processing of an event message by throwing `EventBus.RetryableException`. In API version 43.0 and later, you can test retried event messages by calling `Test.EventBus.deliver()` and inspecting `EventBusSubscriber` fields.

SEE ALSO:


Event and Event Bus Properties in Test Context

In test context, event messages and the event bus have different properties. State information of events and subscribers is reset and is not persisted.

Test Events and the Test Event Bus

When an Apex test publishes an event message, it is published to a test event bus that is separate from the Salesforce event bus. In an Apex test, state information of events and subscribers is reset, as follows.

- The event replay ID value is reset to 0 and starts from 1 for the first test event message.
- Event state information in `EventBusSubscriber` is reset. The last processed replay ID (`EventBusSubscriber.Position`) and the last published replay ID (`EventBusSubscriber.Tip`) are reset to 0.
- When test events are published and processed in subscribers, event state information is updated.
- Subscriber status is reset to `Running` (`EventBusSubscriber.Status`).
- You can query `EventBusSubscriber` to get event state. For example, the following SOQL query gets some information about all trigger subscribers to the `Order_Event__e` event.

```sql
SELECT Name, Position, Retries, LastError
FROM EventBusSubscriber
WHERE Topic='Order_Event__e' AND Type='ApexTrigger'
```

After an Apex test finishes executing, state information of events and subscribers reverts to the non-test values.

Test Events and Limits

Event allocations don't apply to test events, which have their own publishing limit of 500 event messages in a test method. If the number of event messages published from an Apex test context exceeds the limit, an error is returned with the `LIMIT_EXCEEDED` status code. The error is in the `SaveResult` that the `EventBus.publish` Apex method returns.
Testing Event Subscribers

Use an Apex test to test publishing and subscribing to a platform event. When you publish an event message in an Apex test, all event subscribers are notified and start execution, including:

- Apex triggers
- Processes (when using an Apex test class saved with API version 43.0 or later)
- Flows (when using in an Apex test class saved with API version 43.0 or later)

SEE ALSO:
- Event-Driven Software Architecture
- EventBusSubscriber

Deliver Test Event Messages

Deliver test event messages after the Test.stopTest() statement. Alternatively, deliver test event messages at any time with the Test.getEventBus().deliver() method.

Deliver Test Event Messages After Test.stopTest()

To publish platform event messages in an Apex test, enclose the publish statements within Test.startTest() and Test.stopTest() statements. Call the EventBus.publish() method within the Test.startTest() and Test.stopTest() statements. In test context, the EventBus.publish() method enqueues the publish operation. The Test.stopTest() statement causes the event publishing to be carried out and event messages to be delivered to the test event bus. Include your validations after the Test.stopTest() statement. For example, you can validate that a subscribed Apex trigger or a subscribed flow Pause element has performed the expected actions, like creating a Salesforce record.

```
// Create test events
Test.startTest();
// Publish test events with EventBus.publish()
Test.stopTest();
// Perform validations
```

Example: This sample test class contains two test methods. The testValidEvent() method checks that the event was successfully published and fires the associated trigger. The testInvalidEvent() method verifies that publishing an event with a missing required field fails, and no trigger is fired. The testValidEvent() method creates one Low_Ink__e event. After Test.stopTest(), it executes a SOQL query to verify that a case record is created, which means that the trigger was fired. The second test method follows a similar process but for an invalid test.

This example requires that the Low_Ink__e event and the associated trigger are defined in the org.

```
@isTest
class EventTest {
    @isTest static void testValidEvent() {
        // Create a test event instance
        Low_Ink__e inkEvent = new Low_Ink__e{
            Printer_Model__c='MN-123',
            Serial_Number__c='10013',
            Ink_Percentage__c=0.15};
        Test.startTest();
    }
}
```
// Publish test event
Database.SaveResult sr = EventBus.publish(inkEvent);

Test.stopTest();

// Perform validations here

// Verify SaveResult value
System.assertEquals(true, sr.isSuccess());

// Verify that a case was created by a trigger.
List<Case> cases = [SELECT Id FROM Case];
// Validate that this case was found
System.assertEquals(1, cases.size());

@isTest static void testInvalidEvent() {

    // Create a test event instance with invalid data.
    // We assume for this test that the Serial Number__c field is required.
    // Publishing with a missing required field should fail.
    Low_Ink__e inkEvent = new Low_Ink__e(Printer_Model__c='MN-123',
                                         Ink_Percentage__c=0.15);

    Test.startTest();

    // Publish test event
    Database.SaveResult sr = EventBus.publish(inkEvent);

    Test.stopTest();

    // Perform validations here

    // Verify SaveResult value - isSuccess should be false
    System.assertEquals(false, sr.isSuccess());

    // Log the error message
    for(Database.Error err : sr.getErrors()) {
        System.debug('Error returned: ' +
                      err.getStatusCode() +
                      ' - ' +
                      err.getMessage()+ ' - '+err.getFields());
    }

    // Verify that a case was NOT created by a trigger.
    List<Case> cases = [SELECT Id FROM Case];
    // Validate that this case was found
    System.assertEquals(0, cases.size());
}
Deliver Test Event Messages on Demand with Test.getEventBus().deliver()

You can control when test event messages are delivered to subscribers by calling Test.getEventBus().deliver(). Use Test.getEventBus().deliver() to deliver test event messages multiple times and verify that subscribers have processed the test events each step of the way. Delivering event messages multiple times is useful for testing sequential processing of events. For example, you can verify sequential actions of a subscriber in a loop within the same test.

Enclose Test.getEventBus().deliver() within the Test.startTest() and Test.stopTest() statement block.

```java
Test.startTest();
// Create test events
// ...
// Publish test events with EventBus.publish()
// ...
// Deliver test events
Test.getEventBus().deliver();
// Perform validations
// ...
Test.stopTest();
```

**Example:** This test class publishes an Order_Event__e event message and delivers it using Test.getEventBus().deliver(). It verifies that the trigger processed the event message and created a task. A duplicate event message (an event with the same Event_ID__c custom field value) is published and delivered. The test verifies that the trigger didn’t create a task for the duplicate event.

Before you can run this test class, define a platform event with the name of Order_Event__e and the following fields: Event_ID__c of type Text, Order_Number__c of type Text, Has_Shipped__c of type Checkbox.

```java
@isTest
public class MyTestClassDeliver {

    @isTest static void doSomeTesting() {

        Test.startTest();

        // Publish a test event
        Order_Event__e event = new Order_Event__e(
            Event_ID__c='123AB', Order_Number__c='12346', Has_Shipped__c=true);
        Database.SaveResult sr = EventBus.publish(event);

        // Verify that the publish was successful
        System.assertEquals(true, sr.isSuccess());

        // Deliver the test event before Test.stopTest()
        Test.getEventBus().deliver();

        // Check that the case that the trigger created is present.
        List<Task> tasks = [SELECT Id FROM Task];
        // Validate that this task was found.
        // There is only one test task in test context.
        Integer taskCount = tasks.size();
        System.assertEquals(1, taskCount);

        // Publish a duplicate event
        Order_Event__e dupEvent = new Order_Event__e(
```
Event_ID__c='123AB', Order_Number__c='12346', Has_Shipped__c=true);
Database.SaveResult sr2 = EventBus.publish(dupEvent);

// Verify that the publish was successful.
System.assertEquals(true, sr2.isSuccess());
Test.getEventBus().deliver();

// Get all tasks in test context
List<Task> tasksNew = [SELECT Id FROM Task];
// Validate that no task was created and
// the number of tasks should not have changed.
System.assertEquals(taskCount, tasksNew.size());
Test.stopTest();
}
}

This example trigger processes Order_Event__e event messages that the test class publishes.

Note: Because this trigger performs a SOQL query for each event notification received, the Apex governor limit for SOQL queries might be hit.

trigger OrderTrigger on Order_Event__e (after insert) {
    // List to hold all cases to be created.
    List<Task> tasks = new List<Task>();
    // Get user Id for case owner
    User usr = [SELECT Id FROM User WHERE Name='Admin User' LIMIT 1];

    // Iterate through each notification.
    for (Order_Event__e event : Trigger.New) {
        if (event.Has_Shipped__c == true) {
            // Create task only if it doesn't exist yet for the same order
            String eventID = '%' + event.Event_ID__c;
            List<Task> tasksFromQuery =
                [SELECT Id FROM Task WHERE Subject LIKE :eventID];
            if (tasksFromQuery.size() == 0) {
                Task t = new Task();
                t.Priority = 'Medium';
                t.Subject = 'Follow up on shipped order ' + event.Order_Number__c + ' for event ID ' + event.Event_ID__c;
                t.OwnerId = usr.Id;
                tasks.add(t);
            }
        }
    }

    // Insert all tasks in the list.
    if (tasks.size() > 0) {
        insert tasks;
    }
}
Test Retried Event Messages

An Apex trigger can retry processing of an event message by throwing EventBus.RetryableException. In API version 43.0 and later, you can test retried event messages by calling Test.EventBus.deliver() and inspecting EventBusSubscribe fields.

To force redelivery of a retried event message in an Apex test, call Test.EventBus.deliver(). This method also delivers other event messages that have been published after the last deliver() call.

In API version 43.0 or later, you can check these new EventBusSubscriber fields to test retried triggers.

- Retries
- LastError

The EventBusSubscriber.Retries field indicates how many times a trigger was retried.

EventBusSubscriber.LastError indicates the error message that was passed to the throw statement that executed last (throw new EventBus.RetryableException('Error Message')).

**Note:** When EventBus.RetryableException is thrown, EventBusSubscriber.Position isn’t incremented because the trigger didn’t successfully process the event message.

**Example:** This test method delivers a test event message that fires a trigger. The associated event trigger throws EventBus.RetryableException twice. The test verifies that the trigger was retried twice by querying EventBusSubscriber and checking the Retries field value.

Before you can run this test class, define a platform event with the name of Order_Event__e and the following fields: Order_Number__c of type Text and Has_Shipped__c of type Checkbox. This test class assumes there is an associated trigger called OrderTriggerRetry that retries the event. The trigger is not provided in this example.

```java
@isTest
public class MyTestClassRetryDoc {  

@isTest static void doSomeTesting() {  
  Test.startTest();  
  
  // Publish a test event  
  Order_Event__e event = new Order_Event__e(  
    Order_Number__c='12345', Has_Shipped__c=true);  
  Database.SaveResult sr = EventBus.publish(event);  
  // Deliver the initial event message.  
  // This will fire the associated event trigger.  
  Test.getEventBus().deliver();  
  
  // Trigger retries event twice, so loop twice  
  for(Integer i=0;i<2;i++) {  
    // Get info about all subscribers to the event
```
Platform Event Considerations

Learn about special behaviors related to defining, publishing, and subscribing to platform events. Learn how to test platform events. And get an overview of the various events that Salesforce offers.

IN THIS SECTION:

- **Considerations for Defining and Publishing Platform Events**
  Take note of the considerations when defining and publishing platform events.

- **Considerations for Subscribing to Platform Events with Processes and Flows**
  Before you use processes or flows to subscribe to platform events, familiarize yourself with these considerations.

- **Considerations for Publishing and Subscribing to Platform Events with Apex and APIs**
  Before you use Apex or Salesforce APIs to publish and subscribe to platform events, familiarize yourself with these considerations.

- **Decoupled Publishing and Subscription**
  Platform events are published outside of a Lightning Platform database transaction. As a result, the publishing and subscription processes are decoupled—the subscription process can’t assume that an action that the publishing transaction made is committed before an event message is received. Familiarize yourself with some scenarios that can occur from the decoupled behavior.

- **What's the Difference Between the Salesforce Events?**
  Salesforce offers various features that use events, some of which are based on standard platform events. Other features are event-like but aren’t event notifications.
Considerations for Defining and Publishing Platform Events

Considerations for Defining Platform Events

Field-Level Security
All platform event fields are read-only by default, and you can’t restrict access to a particular field. Because platform events aren’t viewable in the Salesforce user interface and aren’t editable, field-level security permissions don’t apply.

Platform Encryption
Platform Encryption is not supported for Platform Event fields.

Enforcement of Field Attributes
Platform event records are validated to ensure that the attributes of their custom fields are enforced. Field attributes include the Required and Default attributes, the precision of number fields, and the maximum length of text fields.

Permanent Deletion of Event Definitions
When you delete an event definition, it’s permanently removed and can’t be restored. Before you delete the event definition, delete the associated triggers. Published events that use the definition are also deleted.

Renaming Event Objects
Before you rename an event, delete the associated triggers. If the event name is modified after clients have subscribed to notifications for this event, the subscribed clients must resubscribe to the updated topic. To resubscribe to the new event, add your trigger for the renamed event object.

No Associated Tab
Platform events don’t have an associated tab because you can’t view event records in the Salesforce user interface.

No SOQL Support
You can’t query event notifications using SOQL.

No Record Page Support in Lightning App Builder
When creating a record page in Lightning App Builder, platform events that you defined show up in the list of objects for the page. However, you can’t create a Lightning record page for platform events because event records aren’t available in the user interface.

Platform Events in Package Uninstall
When uninstalling a package with the option Save a copy of this package’s data for 48 hours after uninstall enabled, platform events aren’t exported.

Event Volume in Package Installations and Upgrades
Installing a managed or unmanaged package that contains a standard-volume platform event causes the event type to be saved as high volume in the subscriber org. Upgrading a managed package doesn’t change the event volume in the subscriber org.

No Support in Professional and Group Editions
Platform events aren’t supported in Professional and Group Edition orgs. Installation of a package that contains platform event objects fails in those orgs.

Considerations for Publishing Platform Events

Publishing Events in Read-Only Mode
During read-only mode, publishing standard-volume platform events results in an exception, and the events aren’t published. Publishing high-volume platform events in read-only mode sometimes fails when the event schema is not up to date in Salesforce. Your org is in read-only mode during Salesforce maintenance activities.
High-Volume Platform Event Persistence
Platform events are temporarily persisted to and served from an industry-standard distributed system during the retention period. A distributed system doesn’t have the same semantics or guarantees as a transactional database. As a result, we can’t provide a synchronous response for an event publish request. Events are queued and buffered, and Salesforce attempts to publish the events asynchronously. In rare cases, the event message might not be persisted in the distributed system during the initial or subsequent publish requests. No mechanism surfaces these types of errors to the publishers or consumers of the event streams.

Considerations for Subscribing to Platform Events with Processes and Flows
Before you use processes or flows to subscribe to platform events, familiarize yourself with these considerations.

Supported Platform Events
Processes and flows can subscribe to custom platform events and these standard platform events.
- BatchApexErrorEvent
- PlatformStatusAlertEvent

Infinite Loops and Limits
Be careful when publishing events from processes or flows because you can get into an infinite loop and exceed limits. For example, a process is associated with the Printer Status platform event. The same process includes an action that creates a Printer Status event message. The process would trigger itself.

To avoid creating an endless loop in an event process, make sure that the new event message’s field values don’t meet the filter criteria for the associated criteria node.

Subscriptions Related List
On the platform event’s detail page, the Subscriptions related list shows which entities are waiting to receive that platform event’s messages. One “Process” subscriber appears in the Subscriptions related list when:
- At least one active event process is associated with that platform event
- At least one flow interview is waiting for that platform event’s messages

Uninstalling Events
Before you uninstall a package that includes a platform event:
- Delete interviews that are waiting for that platform event’s messages
- Deactivate processes that reference the event

Event Processes
These considerations apply only to event processes.

Formulas
Formulas in Process Builder don’t support platform event fields.

Post to Chatter Actions
You can’t include event references in Chatter post messages.

Packaging Event Processes
When you package an event process, the associated object isn’t included automatically. Advise your subscribers to create the object, or manually add the object to your package.

Resumed Flow Interviews
These considerations apply only to flow interviews that resume when a platform event message is received.
Formulas
To reference a platform event in a flow formula, pass the event data into a record variable in the Pause element. Then reference the appropriate field in that record variable.

Event Condition Values
When you filter platform event messages, values for conditions can’t be more than 765 characters.

SEE ALSO:
Decoupled Publishing and Subscription

Considerations for Publishing and Subscribing to Platform Events with Apex and APIs

Before you use Apex or Salesforce APIs to publish and subscribe to platform events, familiarize yourself with these considerations.

Support Only for after insert Triggers
Only after insert triggers are supported for platform events because event notifications can’t be updated. They’re only inserted (published).

Infinite Trigger Loop and Limits
Be careful when publishing events from triggers because you could get into an infinite trigger loop and exceed limits. For example, if you publish an event from a trigger that’s associated with the same event object, the trigger is fired in an infinite loop.

Apex DML Limits for Publishing Events
Each EventBus.publish method call is considered a DML statement, and DML limits apply.

Platform Event Triggers: ownerId Fields of New Records
In platform event triggers, if you create a Salesforce record that contains an ownerId field, set the ownerId field explicitly to the appropriate user. Platform event triggers run under the Automated Process entity. If you don’t set the ownerId field on records that contain this field, the system sets the default value of Automated Process. This example explicitly populates the ownerId field for an opportunity with an ID obtained from another record.

```
Opportunity newOpp = new Opportunity{
    OwnerId = customerOrder.createdById,
    AccountId = acc.Id,
    StageName = 'Qualification',
    Name = 'A ' + customerOrder.Product_Name__c + ' opportunity for ' + acc.name,
    CloseDate = Date.today().addDays(7));
```

For cases and leads, you can alternatively use assignment rules for setting the owner. See AssignmentRuleHeader for the SOAP API or Setting DML Options for Apex.

No Email Support from a Platform Event Trigger
Sending an email message from a platform event trigger using the Messaging.SingleEmailMessage class is not supported. The email can’t be sent because the sender is the Automated Process entity, which has no email address.

API Request Limits for Publishing Events
Because platform events are published by inserting the event sObjects, API request limits apply. For more information, see API Request Limits and Allocations in the Salesforce Limits Quick Reference Guide.

Replaying Past Events
You can replay platform events that were sent in the past 24 hours. You can replay platform events through the API (CometD) but not Apex. The process of replaying platform events is the same as for other Streaming API events. For more information, see the following resources.
Streaming API Developer Guide: Message Durability

Example: Subscribe to and Replay Events Using a Java Client

Example: Subscribe to and Replay Events Using a Visualforce Page

Streaming Replay Client Extensions for Java and JavaScript on GitHub

Note: In rare occasions, some Salesforce maintenance activities, such as an org migration to a new data center, might wipe retained high-volume platform events.

Filtered Subscriptions

Filtered subscriptions in Streaming API aren’t supported for platform events.

Millisecond Time Precision in DateTime Fields

For event messages delivered to CometD clients in JSON format, the DateTime fields include the number of milliseconds. The date format, which is in the ISO 8601 standard, is: `YYYY-MM-DDTHH:mm:ss.sssZ`. In API version 42.0 and earlier, DateTime fields don’t include the millisecond part of the time, and the DateTime format is `YYYY-MM-DDTHH:mm:ssZ`.

For event messages delivered to Apex triggers, DateTime fields don’t include millisecond precision, like Date/Time fields of Salesforce objects.

SEE ALSO:

Platform Event Allocations

Decoupled Publishing and Subscription

Platform events are published outside of a Lightning Platform database transaction. As a result, the publishing and subscription processes are decoupled—the subscription process can’t assume that an action that the publishing transaction made is committed before an event message is received. Familiarize yourself with some scenarios that can occur from the decoupled behavior.

Publisher and Subscriber Processes Not in the Same Transaction

Event publishing and subscription processes don’t run in the same transaction. As a result, a Salesforce record that an event publisher creates after publishing might not be committed to the database before the subscriber receives the event message. If the subscriber looks up the record, it might not be found because it hasn’t been committed yet. For example, consider this scenario.

1. A Process Builder process publishes an event and creates a task record.
2. A trigger on the Task object runs some logic, which delays the commit of the task record.
3. A second Process Builder process, which is subscribed to the event, receives the event and looks up the newly created task. The process returns the following error because the trigger hasn’t finished executing, and the record is not yet committed.

"MyProcess process is configured to start when a MyEvent platform event message occurs. A MyEvent message occurred, but the process didn’t start because no records in your org match the values specified in the process's Object node."

The example uses Process Builder, but the scenario applies to other methods of publishing and subscribing, such as the API and triggers. Conversely, if a subscriber creates a Salesforce record after receiving an event message, the new record might not be found immediately after publishing. The reason is that the event is not processed synchronously after publishing, or the event processing might take a long time if the logic is complex.
Event Publishing and Trigger Order of Execution

If an after insert trigger on a Salesforce object publishes an event, the event can be processed before the Salesforce record in the trigger is committed to the database. For example, consider this scenario.

1. An after insert trigger on a custom object publishes an event message.
2. A Process Builder process is subscribed to the event. Due to the trigger order of execution, the process is fired before the trigger finishes execution and before it commits the new custom object record.
3. The process tries to look up the record to match the event and fails because the record is not found.

One possible solution is to create a scheduled action in your process that publishes a platform event 0 hours after the custom object created date. The scheduled action runs after the custom object is committed. Another option is to publish the event from a @queueable or @future Apex method, which ensures that the publish call is made only when the original transaction is committed. If the trigger’s transaction is rolled back, the @queueable and @future methods aren’t executed and the event isn’t published.

SEE ALSO:
Apex Developer Guide: Triggers and Order of Execution

What’s the Difference Between the Salesforce Events?

Salesforce offers various features that use events, some of which are based on standard platform events. Other features are event-like but aren’t event notifications.

Custom Events

You can use the following types of events to generate and deliver custom messages.

Custom Platform Events

Use custom platform events to deliver secure, scalable, and customizable event notifications within Salesforce or from external sources. Custom platform event fields are defined in Salesforce and determine the data that you send and receive. Apps can publish and subscribe to platform events on the Lightning Platform or in external systems.

Generic Events

Generic events are custom events that contain arbitrary payloads. With a generic event, you can’t define the schema of the event.

Data Events

The following types of events are tied to Salesforce records.

Change Data Capture Events

Salesforce publishes Change Data Capture events for record and field changes.

PushTopic Events

PushTopic events track field changes in Salesforce records and are tied to Salesforce records.

Custom and Data Event Comparison

For a comparison of custom and data events, see Streaming Event Features in the Streaming API Developer Guide.
Standard Events: Security, Apex, and Monitoring

Salesforce publishes the following examples of standard platform events. These predefined events enable monitoring of security-related actions and user actions in Salesforce.

Asset Token Events
Subscribe to an AssetTokenEvent stream to monitor OAuth 2.0 authentication activity. Salesforce publishes an asset token event upon successful completion of an OAuth 2.0 asset token flow for a connected device.

Batch Apex Error Events
Subscribe to a BatchApexErrorEvent stream to catch errors that occur during batch Apex job execution. You can receive all types of errors and exceptions, including uncatchable exceptions, such as Apex limit exceptions.

Event Monitoring Real-Time Events (Pilot)
Event Monitoring Real-Time Events (Pilot) provides standard platform events that you can subscribe to for monitoring user activity in real time, such as logins and running reports. For example, you can subscribe to the event channel for LoginEventStream to receive notifications when users log in.

Note: To enable Real-Time Events (Pilot), contact Salesforce.

Event-Like Features
The following features can trick you into being streaming events, but they’re not.

Event Monitoring Log
Like Real-Time Events (pilot), you can use Event Monitoring to track user activity, such as logins and running reports. Unlike Real-Time Events, Event Monitoring doesn’t send real-time notifications. Instead, it stores user activity in a log that you can query.

Transaction Security Policies
A transaction security policy evaluates user activity, such as logins and data exports, and trigger actions in real time. When a policy is triggered, notifications are sent through email or in-app notifications. You can use standard actions, such as blocking an operation, or custom actions defined in Apex.

Calendar Events
A calendar event is an appointment or meeting that you create and view in the user interface. In SOAP API, the Event object represents a calendar event. These events are calendar items and not notifications that software systems send.

SEE ALSO:
Standard Platform Event Objects

Examples

Check out platform event apps—an end-to-end example using a process and a flow, a Java client, and sample apps that cover business scenarios.

IN THIS SECTION:

End-to-End Example: Printer Supply Automation
This example demonstrates how to make sure that your office printers always have enough paper and ink by using two platform events, a process, and a flow.
Example: Subscribe to and Replay Events Using a Java Client
The Java sample uses a library called Enterprise Messaging Platform (EMP) Connector. EMP Connector is a thin wrapper around the CometD library. It hides the complexity of creating a CometD client and subscribing to Streaming API in Java. The example subscribes to a channel, receives notifications, and supports replaying events with durable streaming.

Platform Event Samples
Check out samples that cover common business scenarios and use platform events along with other Lightning Platform features.

End-to-End Example: Printer Supply Automation
This example demonstrates how to make sure that your office printers always have enough paper and ink by using two platform events, a process, and a flow.

Your company just received a shipment of “smart” printers. You configure the printers to send information to Salesforce once a day. You use that information to update the asset record in Salesforce that represents the printer, then decide whether to order more ink or paper from the vendor. When you do order supplies from the vendor, you schedule a technician to install the new supplies the day after they’re delivered.

IN THIS SECTION:

Platform Events: Printer Status and Vendor Response
This example uses two platform events: one to hold the information coming from the printer (Printer Status) and one to hold the information coming from the vendor (Vendor Response).

Process: Automating Printer Status Events
When Salesforce receives a Printer Status event, a process finds the asset record that’s associated with the printer. It then updates the Total Print Count to match the event. The process evaluates whether the printer has low ink or paper, and if so, launches a flow.

Flow: Automation for Vendor Response Events
The Order Printer Supplies flow starts with a decision that determines whether to order ink or paper. Based on the decision, it submits an order of ink or paper with the vendor by using an Apex action. Then it pauses until the vendor sends a platform event message that says the order has been shipped. When Salesforce receives the specified event, the flow resumes and creates a task for the asset’s owner to install the new supplies.

Platform Events: Printer Status and Vendor Response
This example uses two platform events: one to hold the information coming from the printer (Printer Status) and one to hold the information coming from the vendor (Vendor Response).

The Printer Status platform event includes these custom fields.

<table>
<thead>
<tr>
<th>API Name</th>
<th>Field Label</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial_Number</td>
<td>Serial Number</td>
<td>Text</td>
<td>The printer’s unique identifier. This value is used to locate the corresponding asset record.</td>
</tr>
<tr>
<td>Ink_Status</td>
<td>Ink Status</td>
<td>Text</td>
<td>Values: Full, Medium, Low, or Empty.</td>
</tr>
<tr>
<td>Paper_Level</td>
<td>Paper Level</td>
<td>Number</td>
<td>Paper level in percentage.</td>
</tr>
<tr>
<td>Total_Print_Count</td>
<td>Total Print Count</td>
<td>Number</td>
<td>Aggregate number of pages printed.</td>
</tr>
</tbody>
</table>
The Vendor Response platform event includes these custom fields.

<table>
<thead>
<tr>
<th>API Name</th>
<th>Field Label</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order_Number</td>
<td>Order Number</td>
<td>Text</td>
<td>The order’s unique identifier.</td>
</tr>
<tr>
<td>Expected_Delivery_Date</td>
<td>Expected Delivery Date</td>
<td>Date</td>
<td>The date when the vendor expects the order to be delivered</td>
</tr>
<tr>
<td>Order_Status</td>
<td>Order Status</td>
<td>Text</td>
<td>Values: Ordered, Confirmed, Shipped, Delivered, Delayed, Canceled.</td>
</tr>
</tbody>
</table>

**Process: Automating Printer Status Events**

When Salesforce receives a Printer Status event, a process finds the asset record that’s associated with the printer. It then updates the Total Print Count to match the event. The process evaluates whether the printer has low ink or paper, and if so, launches a flow.

The process starts when it receives a platform event message. The process has three criteria and action groups.
Trigger

The process’s trigger receives a Printer Status event. It uses the serial number to find the asset record that matches the printer.

![Platform Event form]

Criteria 1

The first criteria is set to **No criteria—just execute the actions!** so that it always fires.

The action group contains one immediate action, which updates the asset record’s total print count to match the value from the event.

![Immediate Action form]

After the first criteria’s actions are executed, the process evaluates the next criteria.
Criteria 2

The second criteria checks whether the event’s Ink Level value is set to Low.

The second criteria’s action group contains one immediate action, which launches a flow. The action passes a selection of fields from the asset to the flow that is launched.

<table>
<thead>
<tr>
<th>Flow Variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>assetId</td>
<td>Reference</td>
<td>[Asset].Id</td>
</tr>
<tr>
<td>assetOwner</td>
<td>Reference</td>
<td>[Asset].OwnerId</td>
</tr>
<tr>
<td>inkManufacturer</td>
<td>Reference</td>
<td>[Asset].Ink_Manufacturer__c</td>
</tr>
<tr>
<td>inkNeeded</td>
<td>Boolean</td>
<td>True</td>
</tr>
<tr>
<td>inkType</td>
<td>Reference</td>
<td>[Asset].Ink_Type__c</td>
</tr>
<tr>
<td>paperNeeded</td>
<td>Boolean</td>
<td>False</td>
</tr>
<tr>
<td>paperSize</td>
<td>Reference</td>
<td>[Asset].Paper_Size__c</td>
</tr>
<tr>
<td>serialNumber</td>
<td>Reference</td>
<td>[Asset].SerialNumber</td>
</tr>
</tbody>
</table>

After the second criteria’s actions are executed, the process evaluates the next criteria.

Criteria 3
The third criteria checks whether the event’s Paper Level value is less than 10.

```
<table>
<thead>
<tr>
<th>Source *</th>
<th>Field *</th>
<th>Operator *</th>
<th>Type *</th>
<th>Value *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform event</td>
<td>Paper Level</td>
<td>Less than</td>
<td>Number</td>
<td>10</td>
</tr>
</tbody>
</table>
```

The third criteria’s action group contains one immediate action, which launches the same flow. The action passes a selection of fields from the asset to the flow that’s launched. It passes most of the same values to the flow as the Low Ink criteria group did with two differences: `inkNeeded` is set to false, and `paperNeeded` is set to true.

<table>
<thead>
<tr>
<th>Flow Variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>assetId</td>
<td>Reference</td>
<td>[Asset].Id</td>
</tr>
<tr>
<td>assetOwner</td>
<td>Reference</td>
<td>[Asset].OwnerId</td>
</tr>
<tr>
<td>inkManufacturer</td>
<td>Reference</td>
<td>[Asset].Ink_Manufacturer__c</td>
</tr>
<tr>
<td>inkNeeded</td>
<td>Boolean</td>
<td>False</td>
</tr>
<tr>
<td>inkType</td>
<td>Reference</td>
<td>[Asset].Ink_Type__c</td>
</tr>
<tr>
<td>paperNeeded</td>
<td>Boolean</td>
<td>True</td>
</tr>
<tr>
<td>paperSize</td>
<td>Reference</td>
<td>[Asset].Paper_Size__c</td>
</tr>
<tr>
<td>serialNumber</td>
<td>Reference</td>
<td>[Asset].SerialNumber</td>
</tr>
</tbody>
</table>

The process has only three criteria, so after the third criteria’s actions are executed, the process stops.

**Flow: Automation for Vendor Response Events**

The Order Printer Supplies flow starts with a decision that determines whether to order ink or paper. Based on the decision, it submits an order of ink or paper with the vendor by using an Apex action. Then it pauses until the vendor sends a platform event message that says the order has been shipped. When Salesforce receives the specified event, the flow resumes and creates a task for the asset’s owner to install the new supplies.
Decision Element

The decision includes two outcomes: Ink and Paper. The Ink outcome is true if the variable `{!inkNeeded}` is true. The Paper outcome is true if the variable `{!paperNeeded}` is true.

<table>
<thead>
<tr>
<th>*Label</th>
<th>* Outcome API Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ink</td>
<td>Ink</td>
</tr>
</tbody>
</table>

When to Execute Outcome:
All Conditions Are Met

<table>
<thead>
<tr>
<th>Resource</th>
<th>Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{linkNeeded}</code></td>
<td>Equals</td>
<td><code>{!SGlobalConstant.True}</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>*Label</th>
<th>* Outcome API Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper</td>
<td>Paper</td>
</tr>
</tbody>
</table>

When to Execute Outcome:
All Conditions Are Met

<table>
<thead>
<tr>
<th>Resource</th>
<th>Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{!paperNeeded}</code></td>
<td>Equals</td>
<td><code>{!SGlobalConstant.True}</code></td>
</tr>
</tbody>
</table>
Apex Action Elements

The flow includes two Apex actions that submit a supply order with a vendor but provide different information to it based on whether the flow executed the Ink outcome or Paper outcome. All the variables used for input values (like {!serialNumber} and {!paperSize}) are set when a process launches the flow.

The first Apex action provides information about which ink to order.

![Set Input Values for Ink Action](image)

The second Apex action provides information about which paper to order.

![Set Input Values for Paper Action](image)

In both Apex actions, the action returns an order number. The flow stores that value in the {!orderNumber} variable to reference in the Pause element.

![Set Input Values for Order Number](image)
Pause Element

After the Apex action submits the supply order, the flow waits for confirmation that the order has been shipped. That confirmation is received through the Vendor Response platform event.

The flow pauses until Salesforce receives a Vendor Request event message with specific values. The order number must be the same as the order number that the Apex action provided. And the order status must be Shipped.

```
* Pause Until...
  A Specified Time  A Platform Event Message is Received

* Platform Event
  Vendor Response

Filter Platform Event Messages
Condition Requirements
  All Conditions Are Met

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order_Number__c</td>
<td>{orderNumber}</td>
</tr>
<tr>
<td>Order_Status__c</td>
<td>Shipped</td>
</tr>
</tbody>
</table>
```

When the correct event message is received and the flow resumes, the flow stores the event message’s data in a record variable. That way, you can reference the expected delivery date to calculate when the supplies are scheduled to be installed.

```
Platform Event Message

\{\!vendorResponse\}
```

Create Records Element

When the flow resumes, it creates a task for the asset owner to install the new supplies.
For the task’s field values, the flow uses these resources.

- `{!installDate}`—A formula that calculates the day after the event’s expected delivery date.
- `{!taskDescription}`—A text template that gives more details about the installation.
- `{!assetOwner}`—Provided by the process that launches the flow
- `{!assetId}`—Provided by the process that launches the flow

### Set Field Values for the Task

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivityDate</td>
<td><code>{!installDate}</code></td>
</tr>
<tr>
<td>Description</td>
<td><code>{!taskDescription}</code></td>
</tr>
<tr>
<td>OwnerId</td>
<td><code>{!assetOwner}</code></td>
</tr>
<tr>
<td>Priority</td>
<td>High</td>
</tr>
<tr>
<td>Status</td>
<td>Not Started</td>
</tr>
<tr>
<td>Subject</td>
<td>Install ink on printer</td>
</tr>
<tr>
<td>WhatId</td>
<td>Enter value or search resources...</td>
</tr>
</tbody>
</table>
Example: Subscribe to and Replay Events Using a Java Client

The Java sample uses a library called Enterprise Messaging Platform (EMP) Connector. EMP Connector is a thin wrapper around the CometD library. It hides the complexity of creating a CometD client and subscribing to Streaming API in Java. The example subscribes to a channel, receives notifications, and supports replaying events with durable streaming.

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

EMP Connector is based on Java 8 and uses CometD version 3.1.0. It supports username and password authentication and OAuth bearer token authentication. This walkthrough shows steps only for username and password authentication.

IN THIS SECTION:

Prerequisites
Some tools and a Developer Edition org are required to run the sample.

Define a Custom Platform Event
Before you subscribe to a custom platform event, define the Low_Ink platform event and its fields.

Download and Build the Project
Before you can run the connector examples, download the Java source files and build the Java project.

Subscribe to a Channel and Receive Event Notifications
Use EMP Connector to subscribe to the channel of the Low_Ink__e custom platform event that you defined earlier.

Prerequisites
Some tools and a Developer Edition org are required to run the sample.

- Java Development Kit 8 (see Java Downloads)
- Eclipse IDE for Java Developers (get a recent version from http://www.eclipse.org/downloads/eclipse-packages/)
- To run the tool from the command line: Apache Maven (see https://maven.apache.org/index.html)
- Access to a Developer Edition org
  If you are not already a member of the Lightning Platform developer community, go to developer.salesforce.com/signup and follow the instructions for signing up for a Developer Edition organization. Even if you already have Enterprise Edition, Unlimited Edition, or Performance Edition, use Developer Edition for developing, staging, and testing your solutions against sample data to protect your organization’s live data. This is especially true for applications that insert, update, or delete data (as opposed to simply reading data).

Define a Custom Platform Event
Before you subscribe to a custom platform event, define the Low_Ink platform event and its fields.

1. From Setup, enter Platform Events in the Quick Find box, then select Platform Events.
2. On the Platform Events page, click New Platform Event
3. Complete the standard fields, and optionally add a description.
4. For Event Type, select High Volume.
5. Click Save.
6. To add a field, in the Custom Fields & Relationships related list, click **New**.

7. Create these fields by using the custom field wizard for each field.

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Field Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Printer Model</td>
</tr>
<tr>
<td>Text</td>
<td>Serial Number</td>
</tr>
<tr>
<td>Number (length: 16; decimal places: 2)</td>
<td>Ink Percentage</td>
</tr>
</tbody>
</table>

### Download and Build the Project

Before you can run the connector examples, download the Java source files and build the Java project.

The EMP Connector project includes examples in the GitHub repository’s example folder that use the connector to log in and subscribe to events.

1. To download the project files, do one of the following.
   - Clone the EMP Connector project using git.
     ```
     git clone https://github.com/forcedotcom/EMP-Connector
     ```
   - Download the project zip file from GitHub, and then extract the zip to a local folder.

2. In Eclipse, import the Maven project from the folder where you cloned or extracted the project.
   The dependencies that are specified in the Maven’s `pom.xml` file, such as CometD, are added in the Java project in Eclipse.

3. If the Java project wasn’t automatically built, build it.

   **Note:** If you prefer to run the tool from the command line, generate the JAR file using the Maven command `mvn clean package`. The generated JAR file includes the connector and the `LoginExample` functionality. The JAR file is a shaded JAR—it contains all dependencies for the connector, so you don’t have to download them separately. The JAR file has a `-phat` Maven classifier. You can run the login example from the command line. This command uses the `LoginExample` class by default, which logs in to your production instance.

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

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

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

### Open Source Project

EMP Connector is an open-source project, so you can contribute to it with your own enhancements by submitting pull requests to the repository.
Subscribe to a Channel and Receive Event Notifications

Use EMP Connector to subscribe to the channel of the Low_Ink__e custom platform event that you defined earlier.

1. In the /src/main/java/com/salesforce/emp/connector/example folder, open the LoginExample.java source file.

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

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

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

        long replayFrom = EmpConnector.REPLAY_FROM_EARLIEST;
        if (argv.length == 4) {
            replayFrom = Long.parseLong(argv[3]);
        }

        BearerTokenProvider tokenProvider = new BearerTokenProvider(() -> {
            try {
                return login(argv[0], argv[1]);
            } catch (Exception e) {
                e.printStackTrace(System.err);
                System.exit(1);
                throw new RuntimeException(e);
            }
        });

        BayeuxParameters params = tokenProvider.login();
    }
}
```
2. Run the `LoginExample` class and provide arguments.
   a. In Package Explorer, navigate to the `LoginExample.java` file. Right-click the file, and select Run As > Run Configurations.
   b. On the Arguments tab, add values for the following arguments, separated by a space.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>Your Salesforce username</td>
</tr>
<tr>
<td>password</td>
<td>Your Salesforce password</td>
</tr>
<tr>
<td>channel</td>
<td>The channel name for the event: /event/Low_Ink__e</td>
</tr>
</tbody>
</table>

   c. Click Run.

   The sample is now subscribed to the event channel and is listening to event notifications. As soon as an event notification is generated and received, the tool prints it to the console.

   The sample fetches the earliest saved events within the past 24 hours. Optionally, to receive different events, you can include a replay ID as the last argument. Valid values are:

   - `-1`—Get all new events sent after subscription.
   - `-2`—Get all new events sent after subscription and all past events within the past 24 hours.
   - Specific number—Get all events that occurred after the event with the specified replay ID.

3. To generate an event message for the custom platform event, publish an event message by running Apex in the Developer Console.
   a. In Salesforce Classic, select your name > Developer Console.
   b. In Lightning Experience, click the quick access menu (حرم), and select Developer Console.
   c. In the Developer Console, select Debug > Open Execute Anonymous Window.
   d. In the new window, replace any code with this Apex snippet, which publishes the platform event.

```java
// Create event instance.
Low_Ink__e event = new Low_Ink__e(Printer_Model__c='XZO-5', Serial_Number__c='12345',
   Ink_Percentage__c=0.2);
```
// Publish event.
Database.SaveResult sr = EventBus.publish(event);

// Inspect publishing result for each event
if (sr.isSuccess()) {
    System.debug('Successfully published event.');
} else {
    for(Database.Error err : sr.getErrors()) {
        System.debug('Error returned: ' +
                      err.getStatusCode() +
                      ' - ' +
                      err.getMessage());
    }
}

e. Click Execute. After the platform event is published, EMP Connector receives an event notification, which is printed in the console. The output looks similar to the following.

```
Subscribed: Subscription [/event/Low_Ink__e:-2]
Received:
{"schema":"3l1laWb62nM8omMU0waLdg","payload":{"Serial_Number__c":"12345","CreatedById":"00550000001N45jAAC","CreatedDate":"2018-08-15T21:49:44Z","Ink_Percentage__c":0.2,"Printer_Model__c":"XZO-5"},"event":{"replayId":1}}
```

Note: Generally, do not handle usernames and passwords of others when running code in production. In a production environment, delegate the login to OAuth. The BearerTokenExample.java class uses OAuth authentication.

Platform Event Samples

Check out samples that cover common business scenarios and use platform events along with other Lightning Platform features.

Sample Gallery

The Sample Gallery page includes sample apps that show platform event implementations. The source code of each sample app is provided on GitHub.

Northern Trail Outfitters

This sample app of a fictional retail company demonstrates how to build apps with Aura components and use platform events to integrate with external systems. The app helps Northern Trail Outfitters merchandisers create merchandise mixes for its large retailers. The app submits merchandise mixes to a Heroku-hosted app using platform events.

Pure Aloe

This sample app for a fictional agricultural, manufacturing, and retail company demonstrates how to simplify complex processes and integrate external systems with Aura components, flows, and platform events. The app helps Pure Aloe manage aloe harvest and sell derived aloe products through a distributor channel.

Reference

The reference documentation for platform events covers limits, an API object, and Apex methods.
IN THIS SECTION:

Platform Event Allocations
Learn about the allocations available for platform event definitions, publishing and subscribing to platform events, and event delivery in CometD clients.

EventBusSubscriber
Represents a trigger, process, or flow that is subscribed to a platform event.

EventBus Class
Contains methods for publishing platform events.

Platform Event Error Status Codes
When publishing an event message results in an error, the returned SaveResult contains the error with a status code.

TriggerContext Class
Provides information about the platform event trigger that’s currently executing, such as how many times the trigger was retried due to the EventBus.RetryableException.

Standard Platform Event Objects
Check out the standard platform events that Salesforce publishes.

SEE ALSO:
Salesforce Help: Configure the Process Trigger
Salesforce Help: Flow Element: Pause

Platform Event Allocations
Learn about the allocations available for platform event definitions, publishing and subscribing to platform events, and event delivery in CometD clients.

Common Platform Event Allocations
The following allocations apply to standard-volume and high-volume platform events.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of platform event definitions that can be created in an org</td>
<td>5</td>
<td>5</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>Maximum number of concurrent CometD clients (subscribers) across all channels and for all event types</td>
<td>20</td>
<td>20</td>
<td>2,000</td>
<td>20</td>
</tr>
<tr>
<td>Maximum number of processes that can subscribe to a platform event</td>
<td>5</td>
<td>4,000</td>
<td>4,000</td>
<td>4,000</td>
</tr>
<tr>
<td>Maximum number of active processes that can subscribe to a platform event</td>
<td>5</td>
<td>2,000</td>
<td>2,000</td>
<td>2,000</td>
</tr>
</tbody>
</table>
Note:

- The concurrent client allocation is shared with all types of events that you can subscribe to through Streaming API, including PushTopic, generic, and platform events. A client that exceeds the concurrent client allocation receives an error and can't subscribe. When one of the clients disconnects and a connection is available, the new client can subscribe. See Streaming API Error Codes in the Streaming API Developer Guide.

- Platform events that originate from an installed managed package share the org's allocation for the maximum number of platform event definitions.

### Standard-Volume Platform Event Allocations

The following allocations are for standard-volume events defined in API version 44.0 and earlier.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Delivery: maximum number of delivered event notifications within a 24-hour period, shared by all CometD clients</td>
<td>50,000</td>
<td>25,000</td>
<td>10,000</td>
</tr>
<tr>
<td>Event Publishing: maximum number of event notifications published per hour</td>
<td>100,000</td>
<td>100,000</td>
<td>1,000</td>
</tr>
</tbody>
</table>

If you exceed the event delivery allocation, you receive an error. See Streaming API Error Codes in the Streaming API Developer Guide. Event messages that are generated after exceeding the allocation are stored in the event bus. You can retrieve stored event messages as long as they are within the retention window of 24 hours.

To monitor your standard-volume event delivery usage, use the limits REST API resource and inspect the DailyStandardVolumePlatformEvents value. For more information, see List Organization Limits in the REST API Developer Guide.

1To request a higher number of events delivered to CometD clients, contact Salesforce to purchase an add-on license. The add-on license increases your daily limit of delivered events by 100,000 more events. For example, for Unlimited Edition, the add-on license increases the daily limit of delivered events from 50,000 to 150,000 events. You can purchase multiple add-ons to meet your event requirements for CometD clients. To avoid deployment problems and degradation in service, we recommend that the number of events delivered to CometD clients not exceed 5 million per day. If you require more external events, contact your Salesforce representative to understand how the product can scale to meet your needs.

### High-Volume Platform Event Default Allocations

Each Salesforce edition provides a default allocation for the number of events delivered to CometD clients. The default allocation is enforced daily to ensure fair sharing of resources in the multitenant environment and to protect the service.

The number of delivered events to external CometD clients is counted per subscribed client. If you have multiple client subscribers, your usage is added across all subscribers. For example, you have an Unlimited Edition org with a default allocation of 50,000 events in a 24-hour period. Within a few hours, 20,000 event messages are delivered to two subscribed clients. You have consumed 40,000 events, and you are still entitled to 10,000 events within the 24-hour period.
If you exceed the default event delivery allocation, you receive an error. See Streaming API Error Codes in the Streaming API Developer Guide. Event messages that are generated after exceeding the allocation are stored in the event bus. You can retrieve stored event messages as long as they are within the retention window of 72 hours.

### Table 1: Default Allocations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Delivery: maximum number of delivered event notifications within a 24-hour period, shared by all CometD clients</td>
<td>50,000</td>
<td>25,000</td>
<td>10,000</td>
</tr>
<tr>
<td>Event Publishing: maximum number of event notifications published per hour</td>
<td>250,000</td>
<td>250,000</td>
<td>50,000</td>
</tr>
</tbody>
</table>

### High-Volume Platform Event Add-On License and Usage-Based Entitlements

The add-on increases the 24-hour allocation of delivered event notifications by 100,000 per day (3 million a month) as a usage-based entitlement. The entitlement gives you flexibility in how you use your allocations. The entitlement is not as strictly enforced as the default allocation. With the entitlement, you can exceed your 24-hour event delivery allocation by a certain amount.

Salesforce monitors event overages based on a calendar month, starting with the first day of the month or your contract start date. If you exceed the monthly entitlement, Salesforce contacts you to discuss your event usage needs. The entitlement used for monitoring monthly event overages is the daily allocation multiplied by 30.

When you purchase an add-on license, the hourly event publishing allocation is increased by 25,000 events per hour.

### Table 2: Example: Entitlement with One High-Volume Platform Event Add-On License

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Delivery: entitlement for delivered event notifications within a 24-hour period, shared by all CometD clients</td>
<td>Event Delivery: entitlement for delivered event notifications within a 24-hour period, shared by all CometD clients</td>
<td>Event Delivery: entitlement for delivered event notifications within a 24-hour period, shared by all CometD clients</td>
</tr>
<tr>
<td>You can exceed this entitlement by a certain amount before receiving an error. Salesforce uses the monthly entitlement for event overage monitoring. The monthly entitlement is returned in the <code>limits</code> REST API resource.</td>
<td>You can exceed this entitlement by a certain amount before receiving an error. Salesforce uses the monthly entitlement for event overage monitoring. The monthly entitlement is returned in the <code>limits</code> REST API resource.</td>
<td>You can exceed this entitlement by a certain amount before receiving an error. Salesforce uses the monthly entitlement for event overage monitoring. The monthly entitlement is returned in the <code>limits</code> REST API resource.</td>
</tr>
<tr>
<td>Event Publishing: maximum number of event notifications published per hour</td>
<td>Event Publishing: maximum number of event notifications published per hour</td>
<td>Event Publishing: maximum number of event notifications published per hour</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Delivery: entitlement for delivered event notifications within a 24-hour period, shared by all CometD clients</td>
<td>Event Delivery: entitlement for delivered event notifications within a 24-hour period, shared by all CometD clients</td>
</tr>
<tr>
<td>You can exceed this entitlement by a certain amount before receiving an error. Salesforce uses the monthly entitlement for event overage monitoring. The monthly entitlement is returned in the <code>limits</code> REST API resource.</td>
<td>You can exceed this entitlement by a certain amount before receiving an error. Salesforce uses the monthly entitlement for event overage monitoring. The monthly entitlement is returned in the <code>limits</code> REST API resource.</td>
</tr>
<tr>
<td>Event Publishing: maximum number of event notifications published per hour</td>
<td>Event Publishing: maximum number of event notifications published per hour</td>
</tr>
</tbody>
</table>

The maximum event message size that you can publish is 1 MB. If your event object has hundreds of custom fields or many long text area fields, you could hit this limit. In this case, the publishing call gets an error.
Note: Subscribers on the Lightning Platform (non-CometD clients, including Apex triggers, processes, and flows) don’t count against the 24-hour CometD event delivery limit. The number of event messages that an Apex trigger, process, or flow can process depends on how long the processing takes for each subscriber. The longer the processing time, the longer it takes for the subscriber to reach the tip of the event stream.

Monitor Your High-Volume Event Usage

To monitor your monthly event delivery usage, use the limits REST API resource, and inspect the MonthlyPlatformEvents value. The limits resource returns the monthly maximum allocation. If you don’t have an add-on license, your daily default maximum allocation is the monthly allocation returned divided by 30. For more information, see List Organization Limits in the REST API Developer Guide.

SEE ALSO:

Considerations for Publishing and Subscribing to Platform Events with Apex and APIs

EventBusSubscriber

Represents a trigger, process, or flow that is subscribed to a platform event.

Supported Calls

query()  

Special Access Rules

EventBusSubscriber is read only and can only be queried.

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExternalId</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Properties</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>LastError</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>Properties</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Details</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The error message that the EventBus.RetryableException that the trigger has last thrown contains. This field applies to Apex triggers only. Available in API version 43.0 and later.</td>
</tr>
<tr>
<td>Name</td>
<td><strong>Type</strong> string</td>
</tr>
<tr>
<td></td>
<td><strong>Properties</strong> Filter, Group, Nillable, Sort</td>
</tr>
<tr>
<td></td>
<td><strong>Description</strong> The name of the subscribed item, such as the trigger name. If the subscribed item’s name is “Process”, at least one process or flow Pause element is subscribed to the event.</td>
</tr>
<tr>
<td>Position</td>
<td><strong>Type</strong> int</td>
</tr>
<tr>
<td></td>
<td><strong>Properties</strong> Filter, Group, Nillable, Sort</td>
</tr>
<tr>
<td></td>
<td><strong>Description</strong> The replay ID of the last event that the subscriber processed.</td>
</tr>
<tr>
<td>Retries</td>
<td><strong>Type</strong> int</td>
</tr>
<tr>
<td></td>
<td><strong>Properties</strong> Filter, Group, Nillable, Sort</td>
</tr>
<tr>
<td></td>
<td><strong>Description</strong> The number of times the trigger was retried due to throwing the EventBus.RetryableException. This field applies to Apex triggers only. Available in API version 43.0 and later.</td>
</tr>
<tr>
<td>Status</td>
<td><strong>Type</strong> picklist</td>
</tr>
<tr>
<td></td>
<td><strong>Properties</strong> Filter, Group, Nillable, Restricted picklist, Sort</td>
</tr>
<tr>
<td></td>
<td><strong>Description</strong> Indicates the status of the subscriber. Can be one of the following values:</td>
</tr>
<tr>
<td></td>
<td>• Running—The trigger is actively listening to events.</td>
</tr>
<tr>
<td></td>
<td>• Idle—The trigger hasn’t received events for some time and is not actively listening to events. When new events are sent, the trigger receives the new events after a short delay and switches to the Running state.</td>
</tr>
<tr>
<td></td>
<td>• Error—The trigger has been disconnected and stopped receiving published events. A trigger reaches this state when it exceeds the number of maximum retries with the EventBus.RetryableException. Trigger assertion failures and unhandled events are primarily responsible for the state.</td>
</tr>
</tbody>
</table>
exceptions don’t cause the Error state. To resume trigger execution, fix the trigger code and save it. For a managed package trigger, redeploy the package after fixing the trigger.

- **Suspended**—The trigger is disconnected and can’t receive events due to an internal error. To resume trigger execution, try saving the trigger again. For a managed package trigger, redeploy the package.

### Tip

<table>
<thead>
<tr>
<th>Type</th>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Filter, Group, Nillable, Sort</td>
<td>The replay ID of the last published event.</td>
</tr>
</tbody>
</table>

**Note:** For high-volume platform events, the value for Tip is not available and is always -1.

### Topic

<table>
<thead>
<tr>
<th>Type</th>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Filter, Group, Nillable, Sort</td>
<td>The name of the subscription channel that corresponds to a platform event. The topic name is the event name appended with __e, such as MyEvent__e. The topic is the channel that the subscriber is subscribed to.</td>
</tr>
</tbody>
</table>

### Type

<table>
<thead>
<tr>
<th>Type</th>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Filter, Group, Nillable, Sort</td>
<td>The subscriber type (ApexTrigger). If the subscriber is a process or flow Pause element, the type is blank.</td>
</tr>
</tbody>
</table>

### Usage

Use EventBusSubscriber to query details about subscribers to a platform event. You can get all subscribers for a particular event by filtering on the Topic field, as follows.

```sql
SELECT ExternalId, Name, Position, Status, Tip, Type
FROM EventBusSubscriber
WHERE Topic='Low_Ink__e'
```
EventBus Class
Contains methods for publishing platform events.

Namespace
System

IN THIS SECTION:
EventBus Methods

SEE ALSO:
Platform Events Developer Guide: Publishing Platform Events

EventBus Methods
The following are methods for EventBus. All methods are static.

IN THIS SECTION:
publish(event)
 Publishes the given platform event.
publish(events)
 Publishes the given list of platform events.

**publish(event)**
Publishes the given platform event.

**Signature**
public static Database.SaveResult publish(SObject event)

**Parameters**

*event*
Type: SObject
An instance of a platform event. For example, an instance of MyEvent__e. You must first define your platform event object in your org.

**Return Value**
Type: Database.SaveResult
The result of publishing the given event. Database.SaveResult contains information about whether the operation was successful and the errors encountered. If the isSuccess() method returns true, the event was published for a standard-volume event. For a high-volume event, the publish request is queued in Salesforce and the event message might not be published immediately. For more information, see High-Volume Platform Event Persistence. If isSuccess() returns false, the event publish operation resulted in
errors, which are returned in the `Database.Error` object. This method doesn’t throw an exception due to an unsuccessful publish operation.

`Database.SaveChanges` also contains the `Id` system field. The `Id` field value is not included in the event message delivered to subscribers. It is not used to identify an event message, and is not always unique.

**Usage**

⚠️ **Note:**

- The event insertion occurs non-transactionally. As a result, you can’t roll back published events.
- Apex governor limits apply, including DML limits. Each method execution is counted as one DML statement.

### `publish(events)`

Publishes the given list of platform events.

**Signature**

```java
public static List<Database.SaveChanges> publish(List<SObject> events)
```

**Parameters**

- `events`  
  Type: `List<SObject>`  
  A list of platform event instances. For example, a list of `MyEvent__e` objects. You must first define your platform event object in your org.

**Return Value**

Type: `List<Database.SaveChanges>`

A list of results, each corresponding to the result of publishing one event. For each event, `Database.SaveChanges` contains information about whether the operation was successful and the errors encountered. If the `isSuccess()` method returns `true`, the event was published for a standard-volume event. For a high-volume event, the publish request is queued in Salesforce and the event message might not be published immediately. For more information, see [High-Volume Platform Event Persistence](#). If `isSuccess()` returns `false`, the event publish operation resulted in errors which are returned in the `Database.Error` object. `EventBus.publish()` can publish some passed-in events, even when other events can’t be published due to errors. The `EventBus.publish()` method doesn’t throw exceptions caused by an unsuccessful publish operation. It is similar in behavior to the Apex `Database.insert` method when called with the partial success option.

`Database.SaveChanges` also contains the `Id` system field. The `Id` field value is not included in the event message delivered to subscribers. It is not used to identify an event message, and is not always unique.

**Usage**

⚠️ **Note:**

- The event insertion occurs non-transactionally. As a result, you can’t roll back published events.
- Apex governor limits apply, including DML limits. Each method execution is counted as one DML statement.
Platform Event Error Status Codes

When publishing an event message results in an error, the returned SaveResult contains the error with a status code.

**LIMIT_EXCEEDED**
The number of published platform event messages exceeded the hourly publishing limit or the test limit for event messages published from an Apex test context.

**PLATFORM_EVENT_PUBLISHING_UNAVAILABLE**
Publishing platform event messages failed due to a service being temporarily unavailable. Try again later.

The status code can be returned in Apex in the `Database.SaveResult` in the `Database.Error` object. In SOAP API, it's returned in the `SaveResult` object. In REST API, the status code is returned in the `errors` field in the JSON message.

**TriggerContext Class**

Provides information about the platform event trigger that's currently executing, such as how many times the trigger was retried due to the `EventBus.RetryableException`.

**Namespace**

EventBus

**IN THIS SECTION:**
- TriggerContext Properties
- TriggerContext Methods

**TriggerContext Properties**

The following are properties for `TriggerContext`.

**IN THIS SECTION:**
- `lastError`
  Read-only. The error message that the last thrown `EventBus.RetryableException` contains.
- `retries`
  Read-only. The number of times the trigger was retried due to throwing the `EventBus.RetryableException`.

**lastError**

Read-only. The error message that the last thrown `EventBus.RetryableException` contains.

**Signature**

```java
public String lastError {get;}
```

**Property Value**

Type: String
Usage

The error message that this property returns is the message that was passed in when creating the EventBus.RetryableException exception, as follows.

```java
throw new EventBus.RetryableException(
    'Condition is not met, so retrying the trigger again.'
);
```

retries

Read-only. The number of times the trigger was retried due to throwing the EventBus.RetryableException.

Signature

```java
public Integer retries {get;}
```

Property Value

Type: Integer

TriggerContext Methods

The following are methods for TriggerContext.

IN THIS SECTION:

- **currentContext()**

  Returns an instance of the EventBus.TriggerContext class containing information about the currently executing trigger.

**currentContext()**

Returns an instance of the EventBus.TriggerContext class containing information about the currently executing trigger.

Signature

```java
public static eventbus.TriggerContext currentContext()
```

Return Value

Type: EventBus.TriggerContext

Information about the currently executing trigger.

Standard Platform Event Objects

Check out the standard platform events that Salesforce publishes.
Change Data Capture Events

Salesforce Change Data Capture publishes change events, which represent changes to Salesforce records. Changes include record creation, updates to an existing record, deletion of a record, and undeletion of a record. Change Data Capture events are available since API version 44.0.

Change Event Name

Change events are available for all custom objects and a subset of standard objects. The name of a change event is based on the name of the corresponding object for which it captures the changes.

Standard Object Change Event Name

<Standard_Object_Name>ChangeEvent

Example: AccountChangeEvent

Custom Object Change Event Name

<Custom_Object_Name>__ChangeEvent

Example: Employee__ChangeEvent

Subscription Channels

Subscription channels for change events depend on the name of the change event you want to receive notifications for. Also, a generic channel is provided to receive all notifications.

Channel for All Change Events

To receive event messages for all objects selected for Change Data Capture, use this channel:

/data/ChangeEvents

Standard Object Channel

To receive event messages for changes in a standard object, use this channel:

/data/<Standard_Object_Name>ChangeEvent

Example: AccountChangeEvent

Custom Object Channel

To receive event messages for changes in a custom object, use this channel:

/data/<Custom_Object_Name>__ChangeEvent

Example: Employee__ChangeEvent
Change Event Fields

The record fields in the change event correspond to the fields on the associated Salesforce object or entity that triggered the change. Only new or updated fields are included in the event message.

For example, the fields that can be sent in a change event for the Account object are the Account fields. To look up the fields of a standard object, see Object Reference for Salesforce and Lightning Platform.

Each change event also contains header fields. The header fields are included inside the `ChangeEventHeader` field. They contain information about the event, such as whether the change was an update or delete and the name of the entity, like Account, among other things.

The following example shows the structure of a change event message.

```
{
    "data": {
        "schema": "<schema_ID>",
        "payload": {
            "ChangeEventHeader": {
                "entityName": "...",
                "recordIds": ["..."],
                "changeType": "...",
                "changeOrigin": "...",
                "transactionKey": "...",
                "sequenceNumber": "...",
                "isTransactionEnd": "...",
                "commitTimestamp": "...",
                "commitUser": "...",
                "commitNumber": "...
            },
            "field1": "...",
            "field2": "...",
            ...
        },
        "event": {
            "replayId": <replayID>
        }
    },
    "channel": "/data/<channel>"
}
```

Event Message Example

The following event is sent for a new account.

```
{
    "data": {
        "schema": "TeRuaY6cbI_HsV8Rv1Mc5g",
        "payload": {
            "ChangeEventHeader": {
                "entityName": "Account",
                "recordIds": ["<record_ID>"],
                "changeType": "CREATE",
                "changeOrigin": "com/salesforce/api/soap/44.0;client=Astro",
            }
        },
    },
    "channel": "/data/<channel>"
}
```
"transactionKey": "001b7375-0086-250e-e6ca-b99bc3a8b69f",
"sequenceNumber": 1,
"isTransactionEnd": true,
"commitTimestamp": 1501010206653,
"commitNumber": 92847272780,
"commitUser": "<User_ID>"
},
"Name": "Acme",
"Description": "Everyone is talking about the cloud. But what does it mean?",
"OwnerId": "<Owner_ID>",
"CreatedDate": "2017-07-25T19:16:44Z",
"CreatedById": "<User_ID>",
"LastModifiedDate": "2017-07-25T19:16:44Z",
"LastModifiedById": "<User_ID>"
},
"event": {
  "replayId": 6
},
"channel": "/data/ChangeEvents"

Resources
For more information about Change Data Capture, see Change Data Capture Developer Guide.

Standard Platform Event Object List
Salesforce publishes standard platform events to monitor various activities in the org, such as security actions or batch Apex errors.

AssetTokenEvent
Represents a standard platform event associated with an asset token, such as token issuance and registration of a connected device as an Asset.

BatchApexErrorEvent
Represents a standard platform event associated with a batch Apex class.

PlatformStatusAlertEvent
Represents a standard platform event associated with an alert. The alert occurs during the processing of a user request or service job execution for Einstein Next Best Action.