Platform Events Developer Guide

Version 41.0, Winter '18

18

@salesforcedocs

Last updated: December 8, 2017

CONTENTS

CHAPTER 1 Delivering Custom Notifications with Platform Events

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 Force.com platform can produce or consume.

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.

EDITIONS

Available in: both Salesforce Classic and Lightning Experience

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

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.

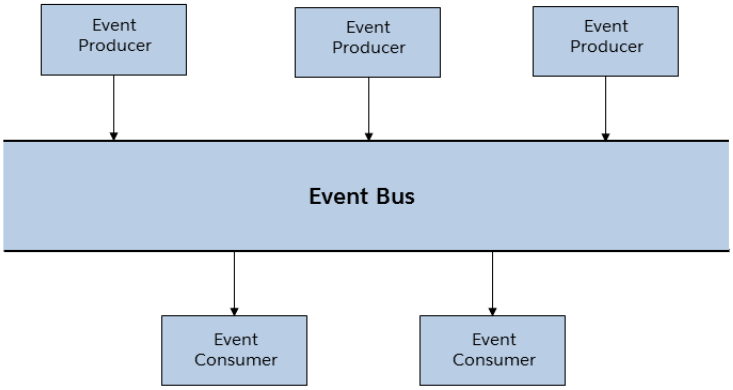
Channel
: A conduit in which an event producer transmits a message. Event consumers subscribe to the channel to receive messages. Also referred to as event bus in Salesforce.

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.

Platform events are defined in Salesforce in the same way that you define custom objects. Create a platform event definition by giving it a name and adding custom fields. The custom fields define the data that is sent in the event messages. 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.

Field Name	Field API Name	Field Type
Printer Model	Printer_Model__c	Text
Serial Number	Serial_Number__c	Text
Ink Percentage	Ink_Percentage__c	Number

You can publish and consume platform events by using Apex or an API. Platform events integrate with the Salesforce platform through Apex triggers. Triggers are the event consumers on the Salesforce platform that listen to event messages. When an external app through the API or a native Force.com app through Apex publishes the event message, a trigger on that event is fired. Triggers run the actions in response to the event notifications. 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.

External apps can listen to event messages by subscribing to a channel through CometD. Platform apps, such as Visualforce pages and Lightning components, can subscribe to event messages with CometD as well.

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 object. Unlike custom 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 Force.com platform. As a result, publishing 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.

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

CHAPTER 2 Defining Your Platform Event

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

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 lifecycle.

EDITIONS

Available in: both Salesforce Classic and Lightning Experience

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

USER PERMISSIONS

To create and edit platform event definitions:
- Customize Application

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**.
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 **Standard Volume**.

   Note: If you are enrolled in the High-Volume Platform Events pilot, the Event Type dropdown also displays the High Volume option for creating high-volume events. For more information about the pilot, see the Salesforce Winter '18 Release Notes.
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
`Label`	Name used to refer to your platform event in a user interface page.
`Plural Label`	Plural name of the platform event.

Field	Description
`Starts with a vowel sound`	If it's appropriate for your org's default language, indicate whether the label is preceded by "an" instead of "a."
`Object Name`	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.
`Description`	Optional description of the object. A meaningful description helps you remember the differences between your events when you are viewing them in a list.
`Deployment Status`	Indicates whether the platform event is visible to other users.

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)

`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, 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 Platform Events

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

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 lifecycle.

Example: The CustomObject metadata type represents platform events.

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.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<CustomObject xmlns="http://soap.sforce.com/2006/04/metadata">
    <deploymentStatus>Deployed</deploymentStatus>
    <eventType>StandardVolume</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>
        <isNameField>false</isNameField>
        <isSortingDisabled>false</isSortingDisabled>
        <label>Serial Number</label>
        <length>20</length>
        <required>false</required>
        <type>Text</type>
        <unique>false</unique>
    </fields>
    <label>Low Ink</label>
    <pluralLabel>Low Ink</pluralLabel>
</CustomObject>
```

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

```
<?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>41.0</version>
</Package>
```

Retrieve 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 version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
    <types>
        <members>*</members>
        <name>CustomObject</name>
    </types>
    <version>41.0</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*.

CHAPTER 3 Publishing Platform Events

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

Publish Event Messages with Processes

Use Process Builder to publish event messages from a Force.com app as part of an automated process.

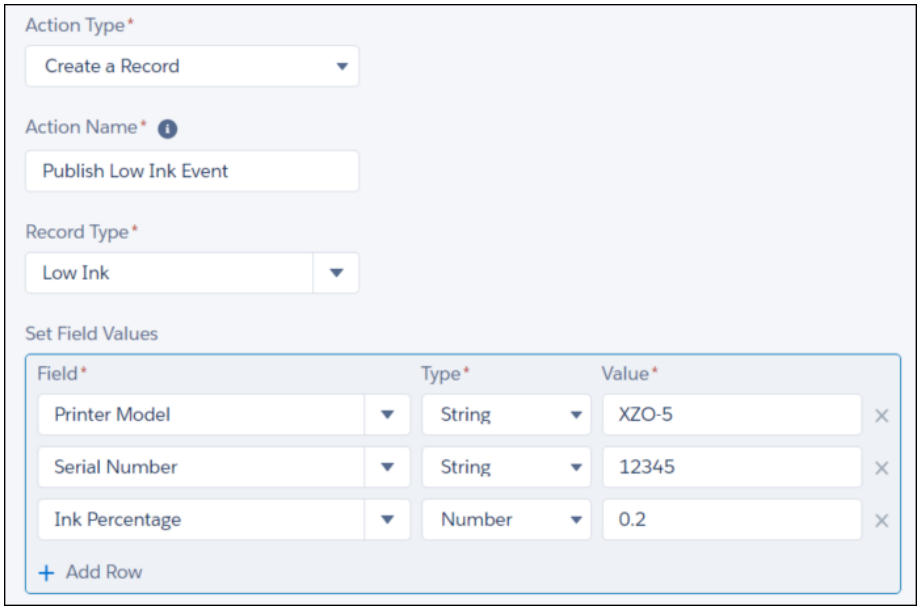
To publish event messages, add a Create a Record action to the appropriate process. Where you'd usually pick an object to create, select the platform event.

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.

Field	Type	Value
Printer Model	String	XZO-5
Serial Number	String	12345
Ink Percentage	Number	0.2



3. Save the action and activate the process.

SEE ALSO:

Salesforce Help: Lightning Process Builder

Publish Event Messages with Flows

Use Visual Workflow to publish event messages from a Force.com app as part of some user interaction, an automated process, Apex, or workflow action.

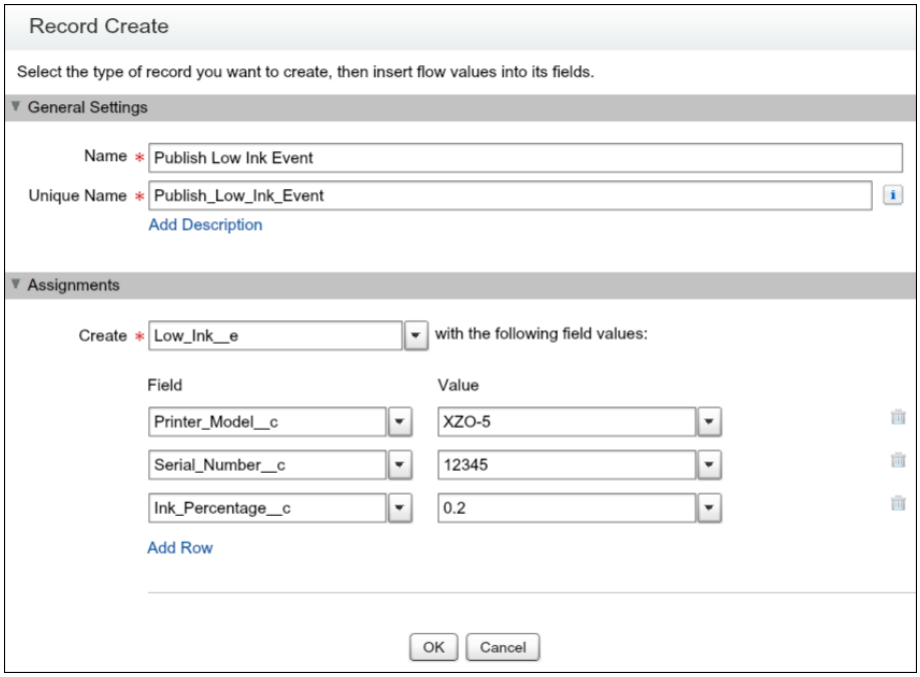
To publish event messages, add a Record Create or a Fast Create element to the appropriate flow. Where you'd usually pick an object to create, select the platform event.

For example, here's how to configure a Record Create element 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 Create, enter `low` and select **Low_Ink__e**.
2. Add these fields.

Field	Value
Printer Model	XZO-5
Serial Number	12345
Ink Percentage	0.2



3. Save and activate the flow.

SEE ALSO:

Visual Workflow Guide

Publish Event Messages with Apex

Use Apex to publish event messages from a Force.com 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. The example assumes that the `Low Ink` platform event is defined in your org.

```
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. Otherwise, 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.

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

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.

REST API

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

```
/services/data/v41.0/sobjects/Event_Name__e/
```

Example: If you've defined a platform event named `Low Ink`, publish event notifications by inserting `Low_Ink__e` records. This example creates one event of type `Low_Ink__e` in REST API.

REST endpoint:

```
/services/data/v41.0/sobjects/Low_Ink__e/
```

Request body:

```
{
   "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 version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
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!AQMAQOshATCQ4fBaYFOTrHVixfEO6l...</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>
            <ns1:fieldsToNull xsi:nil="true"/>
            <ns1:Id xsi:nil="true"/>
            <Printer_Model__c>XZO-600</Printer_Model__c>
        </ns2:sObjects>
        <ns2:sObjects>
            <ns1:type>Low_Ink__e</ns1:type>
            <ns1:fieldsToNull xsi:nil="true"/>
            <ns1:Id xsi:nil="true"/>
            <Printer_Model__c>XYZ-100</Printer_Model__c>
        </ns2:sObjects>
        <ns2:sObjects>
            <ns1:type>Low_Ink__e</ns1:type>
            <ns1:fieldsToNull xsi:nil="true"/>
            <ns1:Id xsi:nil="true"/>
            <Printer_Model__c>XYZ-9000</Printer_Model__c>
        </ns2:sObjects>
    </ns2:create>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

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

```
<?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:

Force.com REST API Developer Guide

SOAP API Developer Guide: `create()` call

Bulk API Developer Guide

CHAPTER 4 Subscribing to Platform Events

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

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:

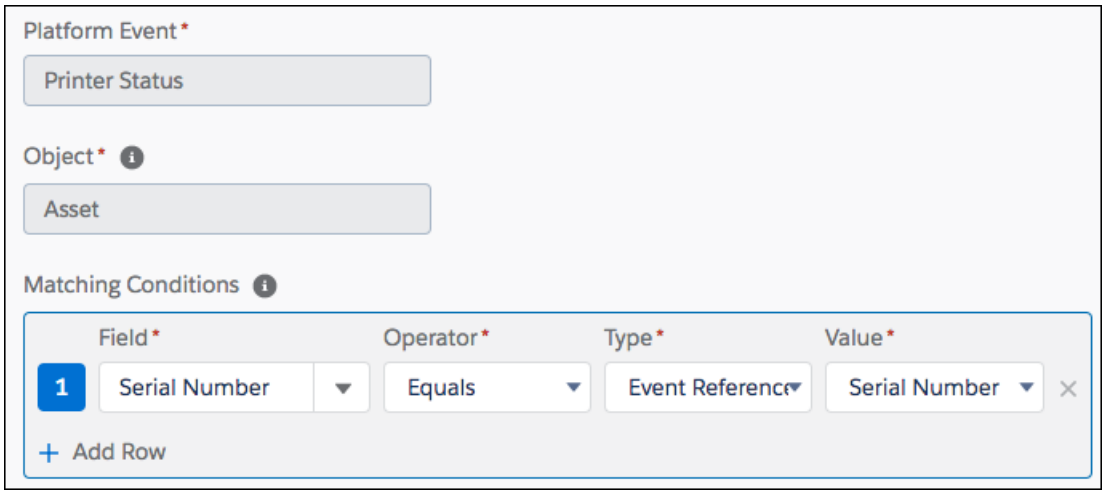
Salesforce Help: Set Up Debug Logging

Subscribe to Platform Event Notifications with Processes

Use processes to subscribe to events. You can receive event notifications in processes regardless of how they were published—through Apex, APIs, flows, or other processes. Processes provide an autosubscription mechanism.

To subscribe to event notifications, create a process that starts when a platform event occurs. In the process's trigger, associate the process with a platform event type and an object type.

Example: This process starts when a Printer Status event occurs. When it starts, the process looks for an Asset record whose serial number matches the event notification's serial number.



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

A process evaluates platform event notifications in the order they're received. The order of events is based on the event replay ID. A process 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 record change processes, event processes don't execute in the same Apex transaction as whatever published the event. The process runs asynchronously under the Automated Process entity. As a result, there can be a delay between when an event is published and when the process evaluates the event. 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.

Event processes and record change processes have similar limitations. For example, they're both subject to Apex governor limits.

SEE ALSO:

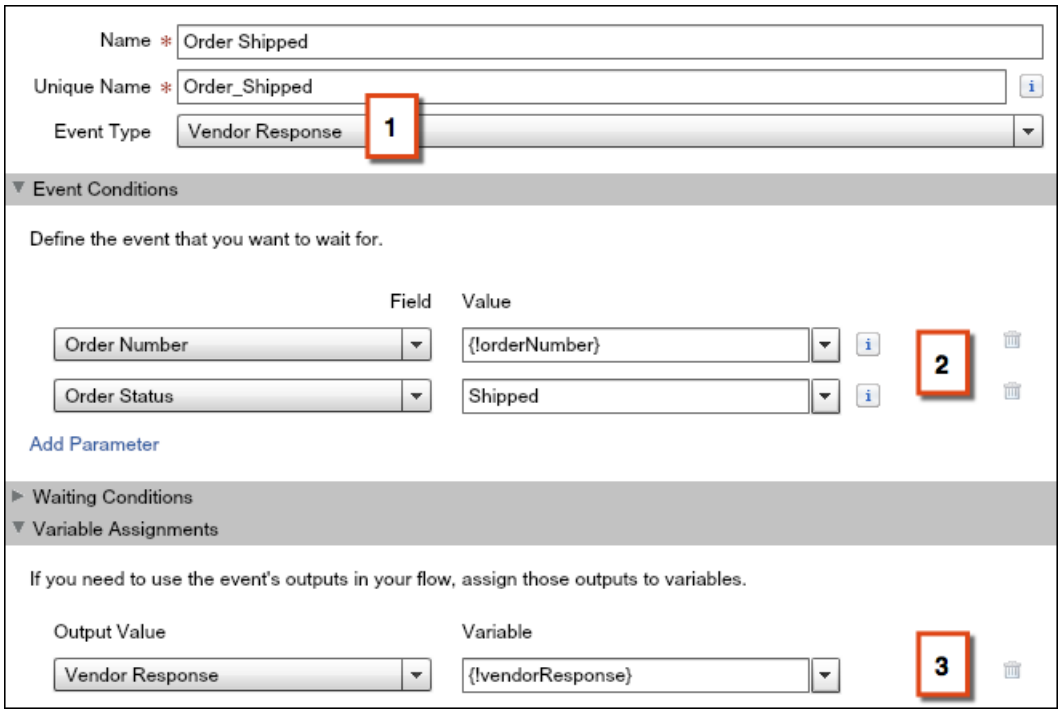
- Considerations for Subscribing to Platform Events with Processes and Flows
- *Salesforce Help*: Process Limits and Considerations
- Set Up Debug Logs for Event Subscriptions
- Obtain Processes That Subscribe to a Platform Event

Subscribe to Platform Event Notifications with Flows

Use flows to subscribe to events. You can receive event notifications in flows regardless of how they were published—through Apex, APIs, flows, or other processes. Flows provide an autosubscription mechanism.

To subscribe to event notifications, create a flow that waits for a platform event to occur. In the Wait element, select the event type to wait for and identify the values the event must have for the flow to resume. Optionally, create an sObject variable to store the event's data when the flow resumes.

Example: The wait event resumes the flow when a vendor response event occurs (1). The event's order number must match the {!orderNumber} variable value, and the order status must be Shipped (2). When the flow resumes, the {!vendorResponse} sObject variable gets populated with the event's data (3).



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

Flow interviews evaluate platform event notifications in the order they're received. The order of events is based on the event replay ID. A flow interview 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.

Resumed flow interviews execute in a separate Apex transaction than the transaction that published the event. The flow interview resumes asynchronously under the Automated Process entity. As a result, there can be a delay between when an event is published and when the interview evaluates the event. 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. System fields, such as `CreatedById` and `LastModifiedById`, reference the user who published the event.

SEE ALSO:

- Considerations for Subscribing to Platform Events with Processes and Flows
- *Visual Workflow Guide*: Limits and Considerations for Visual Workflow
- *Visual Workflow Guide*: Limitations for Waiting Flows

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.

```
// 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
    User usr = [SELECT Id FROM User WHERE Name='Admin User' LIMIT 1];

    // 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;
            cs.OwnerId = usr.Id;
            cases.add(cs);
        }
    }

    // Insert all cases corresponding to events received.
    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 API.

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.

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.

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 API

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.

You can run a trigger up to 10 times when it is retried (the initial run and 9 retries). After the trigger is retried 9 times, it moves to the error state and stops processing new events. To resume event processing, fix the trigger and save it. Events sent after the trigger moves to the error state and before it returns to the running state are not resent to the trigger. We recommend setting a limit to retry the trigger to less than 9 times. Use the `EventBus.TriggerContext.currentContext().retries` property to check how many times the trigger has been retried.

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.

```
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.
        }
    }
}
```

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. Here is the format of the platform event topic (channel) name.

```
/event/Event_Name__e
```

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

```
/cometd/41.0
```

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.

```
{
  "data": {
    "schema": "dffQ2QLzDNHqwB8_sHMxdA",
    "payload": {
      "CreatedDate": "2017-04-09T18:31:40Z",
      "CreatedById": "005D0000001cSZs",
      "Printer_Model__c": "XZO-5",
      "Serial_Number__c": "12345",
      "Ink_Percentage__c": 0.2
    },
    "event": {
      "replayId": 2
    }
  },
  "channel": "/event/Low_Ink__e"
}
```

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: `/v`***`XX.X`***`/event/eventSchema/`***`Schema_ID`***. Alternatively, you can retrieve the event schema by supplying the event name to this endpoint: `/v`***`XX.X`***`/sobjects/`***`Platform_Event_Name__e`***`/eventSchema`. For more information, see:

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

You can use EMP Connector to receive delivered events. The connector subscribes to streaming events and platform events—only the supplied topic name is different. See Example: Subscribe to and Replay Events Using a Java Client in the Streaming API Developer Guide. For the *`topic`* argument, provide `/event/Low_Ink__e`. The topic name value is based on the example event `Low Ink`.

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:

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

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 and flow interview subscribers to a platform event channel aren't exposed in the user interface or the API.

Obtain Processes That Subscribe to a Platform Event

Event processes aren't reflected 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 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 version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
   <types>
      <members>EventSubscription</members>
      <name>*</name>
   </types>
```

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 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 and the event last published. 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.

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.
- `Suspended`—The trigger is disconnected and can't receive events due to lack of permissions.
- `Expired`—The trigger's connection expired. In rare cases, subscriptions can expire if they're inactive for an extended period of time.

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

CHAPTER 5 Platform Event Considerations and Testing

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.

Considerations for Defining Platform Events

Take note of the considerations when 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.

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 Subscribing to Platform Events with Processes and Flows

Before you use processes or flows to subscribe to platform events, familiarize yourself with these considerations.

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 notification. 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 for notifications of that platform event to occur. 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 to occur

Uninstalling Events
Before you uninstall a package that includes a platform event:

- Delete interviews that are waiting for the event to occur
- Deactivate processes that reference the event

Testing Events
Event processes or Wait elements subscribed to platform events don't support Apex tests.

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.

Scheduled Process Actions
Scheduled actions aren't supported in event processes.

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 occurs.

Formulas
To reference a platform event field in a flow, pass the event data into an sObject variable in the Wait element. Then reference the appropriate field in that sObject variable.

Event Condition Values
In flow event conditions, values can't be more than 765 characters.

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

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.

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 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 maintenance activities on the Salesforce application servers wipe retained events. For example, maintenance activities, such as disaster recovery of servers after unexpected outages, a site switch of a Salesforce instance, or an org migration to a new data center, can wipe retained events. To prevent losing events, receive only the latest events by subscribing to the tip of the queue (-1 replay option).

Filtered Subscriptions

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

Publishing Events in Read-Only Mode

During read-only mode, publishing platform events results in an exception and the events aren't published. In contrast, you can publish high-volume platform events in read-only mode. Your org is in read-only mode during Salesforce maintenance activities.

SEE ALSO:

Platform Event Limits

Test Your Platform Event Trigger in Apex

Ensure that your platform event trigger is working properly by adding an Apex test. Before you can package or deploy any Apex code (including triggers) to production, your Apex code must have tests and sufficient code coverage. To publish platform events in an Apex test, enclose the publish statements within `Test.startTest()` and `Test.stopTest()` statements.

Call the `publish` method within the `Test.startTest()` and `Test.stopTest()` statements. In test context, the `publish` method enqueues the publish operation. The `Test.stopTest()` statement causes the event publishing to be carried out. Include your validations after the `Test.stopTest()` statement.

```
// Create test events
Test.startTest();
// Publish test events
Test.stopTest();
// Perform validation here
```

You can publish up to 500 events in a test method.

Example: This sample test class contains two test methods. The `testValidEvent` method checks that publishing an event is successful 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 is defined in the org.

```
@isTest
public 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());
    }
}
```

SEE ALSO:

Apex Developer Guide: Testing and Code Coverage

What Is the Difference Between the Salesforce Events?

Salesforce offers various features that use events. Except for Platform Events and Streaming API generic events, most of these events are notifications within Salesforce or calendar items.

The following is a partial list of the types of events provided.

Platform Events

Platform events enable you to deliver secure, scalable, and customizable event notifications within Salesforce or from external sources. 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 Force.com Platform using Apex or in external systems using CometD.

Streaming API Events

Streaming API provides two types of events that you can publish and subscribe to: PushTopic and generic. PushTopic events track field changes in Salesforce records and are tied to Salesforce records. Generic events contain arbitrary payloads. Both event types don't provide the level of granular customization that platform events offer. You can send a custom payload with a generic event, but you can't define the data as fields. You can't define those types of events in Salesforce, and you can't use them in Apex triggers.

Event Monitoring

Event monitoring enables admins to track user activity and the org's performance. In this context, events are actions that users perform, such as logins and exporting reports. The events are internal and logged by Salesforce. You can query the events, but you can't publish the events or subscribe to them in real time.

Transaction Security Policies

Transaction security policies evaluate 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. Actions can be standard actions, such as blocking an operation, or a custom action defined in Apex.

Calendar Events

Calendar events in Salesforce are appointments and meetings you can create and view in the user interface. In the SOAP API, the Event object represents a calendar event. Those events are calendar items and not notifications that software systems send.

This guide focuses on Platform Events only.

CHAPTER 6 End-to-End Example: Printer Supply Automation

In this chapter ...

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.

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.

Unique Name	Label	Data Type	Description
Serial_Number	Serial Number	Text	The printer's unique identifier. This value is used to locate the corresponding asset record.
Ink_Status	Ink Status	Text	Values: Full, Medium, Low, or Empty.
Paper_Level	Paper Level	Number	Paper level in percentage.
Total_Print_Count	Total Print Count	Number	Aggregate number of pages printed.

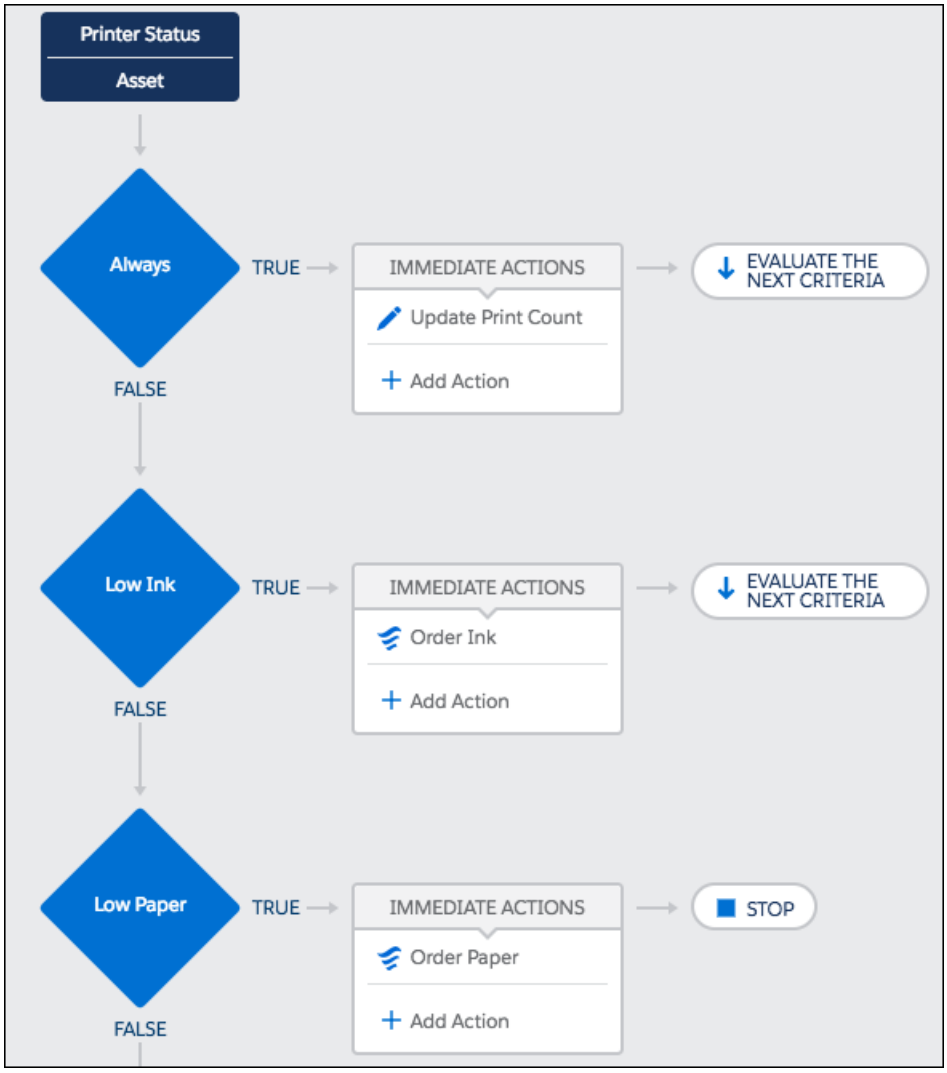
The Vendor Response platform event includes these custom fields.

Unique Name	Label	Data Type	Description
Order_Number	Order Number	Text	The order's unique identifier.
Expected_Delivery_Date	Expected Delivery Date	Date	The date when the vendor expects the order to be delivered
Order_Status	Order Status	Text	Values: Ordered, Confirmed, Shipped, Delivered, Delayed, Canceled.

Process: Automating Printer Status Events

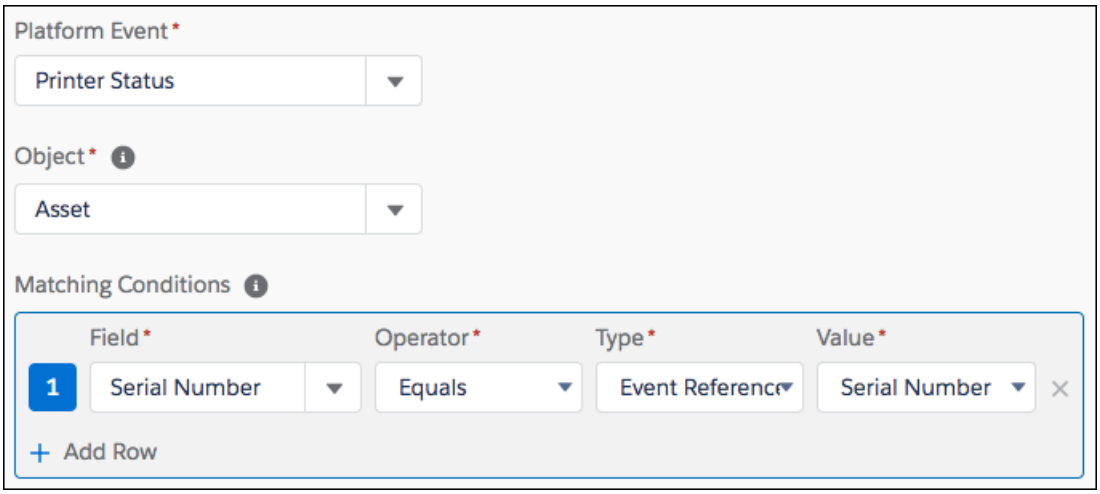
When Salesforce receives a Printer Status event, a process finds the asset record that's associated with the printer. It then updates the Total Print Count to match the event. The process evaluates whether the printer has low ink or paper, and if so, launches a flow.

The process starts when a platform event occurs and has three criteria and action groups.

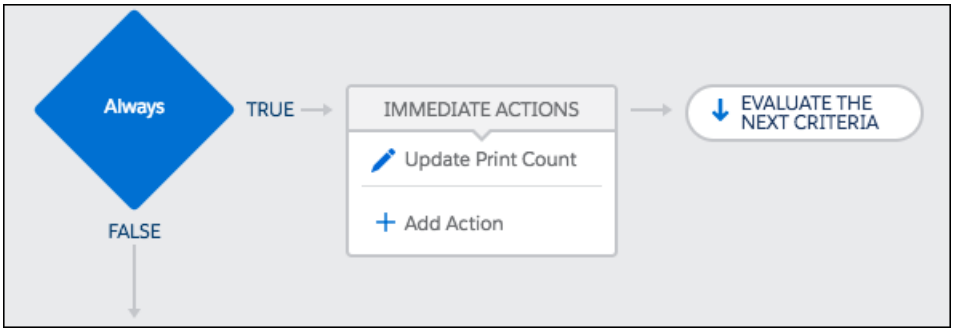


Trigger

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

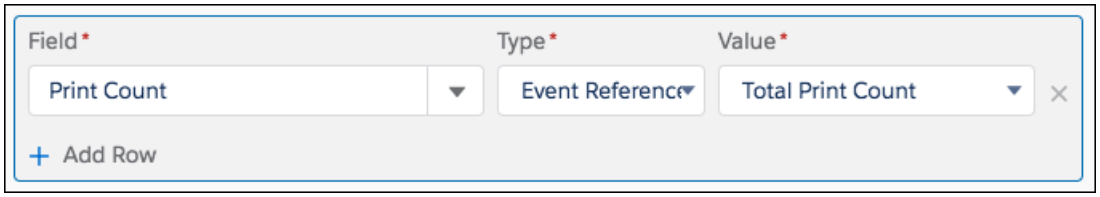


Criteria 1



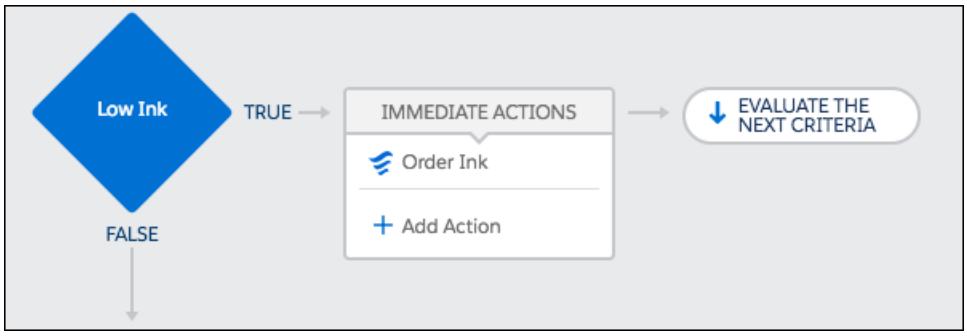
The first criteria is set to **No criteria–just execute the actions!** so that it always fires.

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

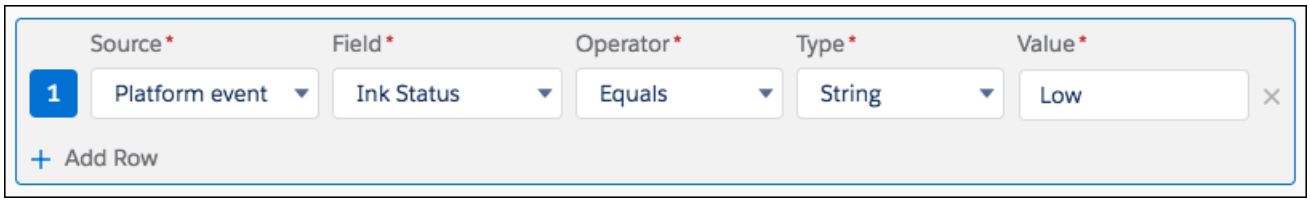


After the first criteria's actions are executed, the process evaluates the next criteria.

Criteria 2



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

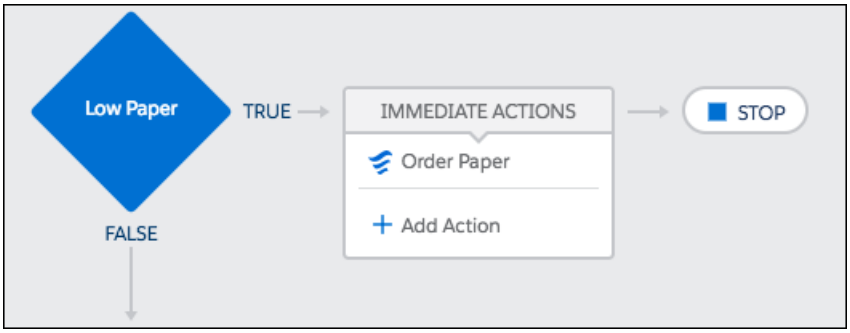


The second criteria's action group contains one immediate action, which launches a flow. The action passes a selection of fields from the asset to the flow that's launched.

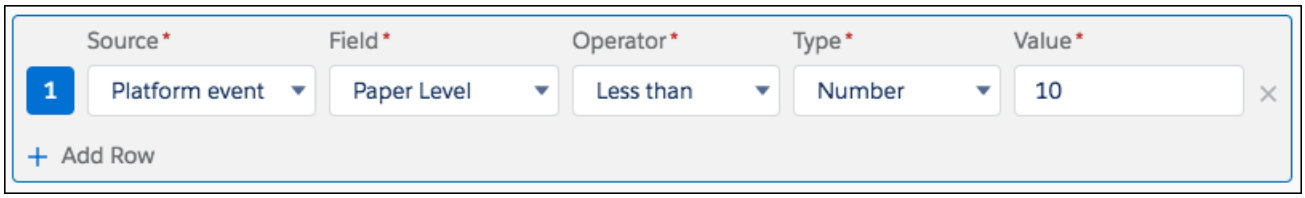
Flow Variable	Type	Value
assetId	Reference	[Asset].Id
assetOwner	Reference	[Asset].OwnerId
inkManufacturer	Reference	[Asset].Ink_Manufacturer__c
inkNeeded	Boolean	True
inkType	Reference	[Asset].Ink_Type__c
paperNeeded	Boolean	False
paperSize	Reference	[Asset].Paper_Size__c
serialNumber	Reference	[Asset].SerialNumber

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.



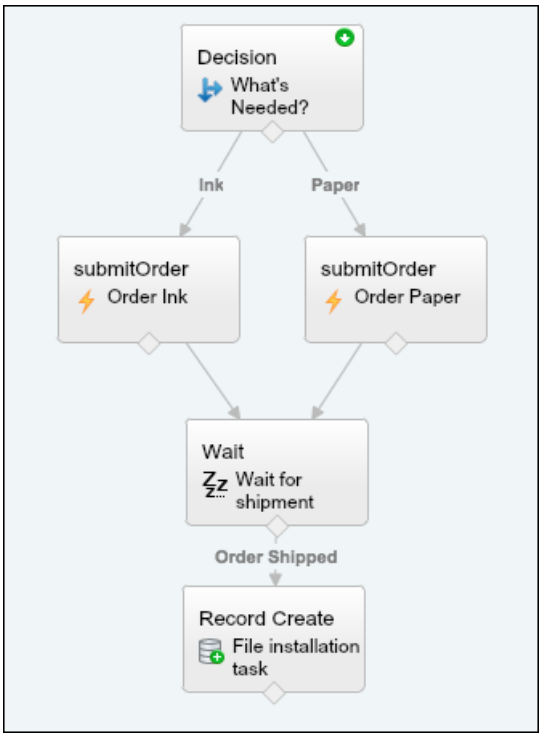
The third criteria's action group contains one immediate action, which launches the same flow. The action passes a selection of fields from the asset to the flow that's launched. It passes most of the same values to the flow as the Low Ink criteria group did with two differences: `inkNeeded` is set to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

Flow Variable	Type	Value
assetId	Reference	[Asset].Id
assetOwner	Reference	[Asset].OwnerId
inkManufacturer	Reference	[Asset].Ink_Manufacturer__c
inkNeeded	Boolean	False
inkType	Reference	[Asset].Ink_Type__c
paperNeeded	Boolean	True
paperSize	Reference	[Asset].Paper_Size__c
serialNumber	Reference	[Asset].SerialNumber

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

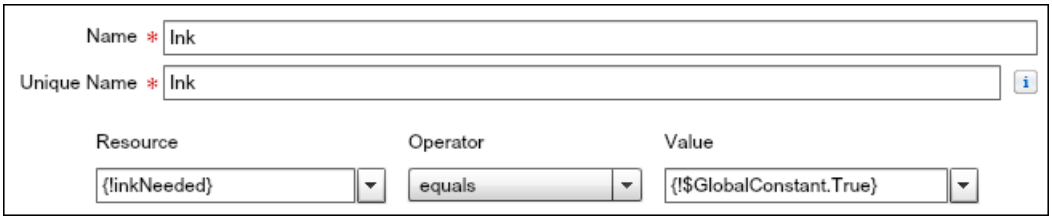
Flow: Automation for Vendor Response Events

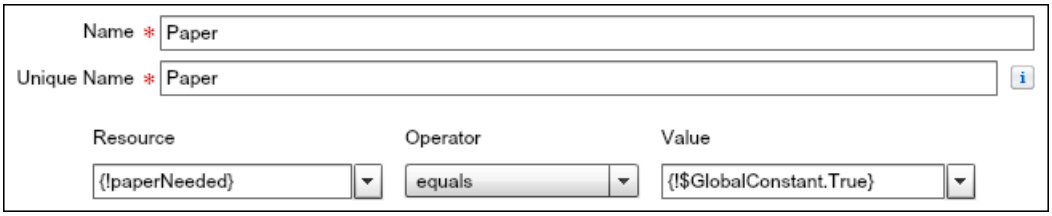
The Order Printer Supplies flow starts with a decision that determines whether to order ink or paper. Based on the decision, it invokes Apex code to submit an order of ink or paper with the vendor. Then it waits for the vendor to send a platform event of type Vendor Response that says the order has been 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.

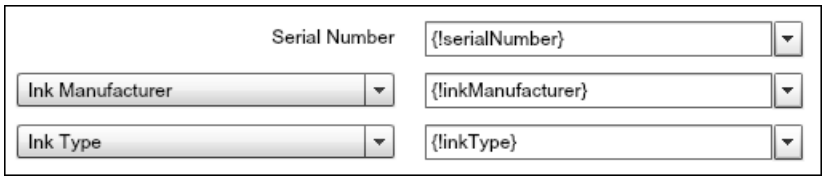




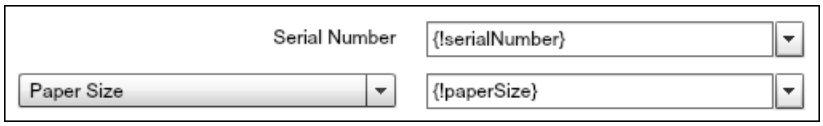
Apex Elements

The flow includes two Apex elements 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 element provides information about which ink to order.



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



In both Apex elements, after the class submits the order, it returns an order number. The flow stores that value in the `{!orderNumber}` variable to use in the Wait element.

Wait Element

After the Apex class 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 waits for a specific Vendor Response. The order number must be the same as the order number that the Apex class provided. And the order status must be Shipped.

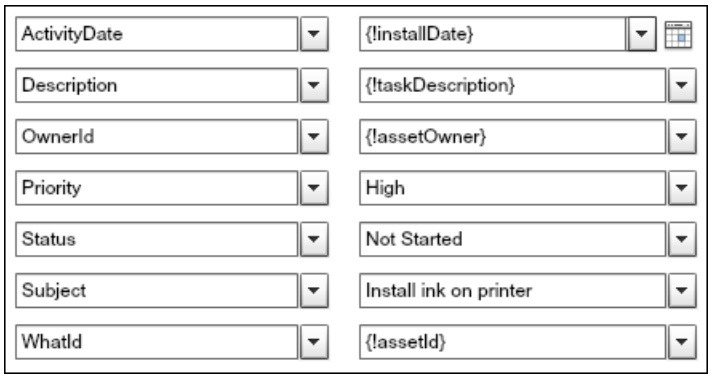
When the correct event occurs and the flow resumes, the flow stores the event's data in an sObject variable. That way, you can reference the expected delivery date to calculate when the supplies are scheduled to be installed.

Record Create 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



CHAPTER 7 Reference

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

Platform Event Limits
Limits are enforced for publishing platform events, event delivery in CometD clients, and platform event definitions.

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

EventBus Class
Contains methods for publishing platform events.

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

SEE ALSO:

Salesforce Help: Configure the Process Trigger

Visual Workflow Guide: Flow Wait Element

Platform Event Limits

Limits are enforced for publishing platform events, event delivery in CometD clients, and platform event definitions.

Description	Performance and Unlimited Editions	Enterprise Edition	All other supported editions
Maximum number of event notifications published per hour	100,000	100,000	1,000
Maximum number of event notifications delivered to CometD clients within a 24-hour period[1]	50,000	25,000	10,000
Maximum number of platform event definitions that can be created in an org	100	50	5
Maximum number of processes or flow interviews that can subscribe to a platform event	500	500	500
Maximum number of active processes or flow interviews that can subscribe to a platform event	50	50	50

[1]To 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 additional external events, contact your Salesforce representative to understand how the product can scale to meet your needs.

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

Field	Details
`ExternalId`	**Type** string **Properties** Filter, Group, Nillable, Sort **Description** The ID of the subscriber. For example, the trigger ID.
`Name`	**Type** string **Properties** Filter, Group, Nillable, Sort **Description** 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 Wait element is subscribed to the event.
`Position`	**Type** int **Properties** Filter, Group, Nillable, Sort **Description** The replay ID of the last event that the subscriber processed.

Field	Details
`Status`	**Type** picklist **Properties** Filter, Group, Nillable, Restricted picklist, Sort **Description** Indicates the status of the subscriber. Can be one of the following values: • `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. • `Suspended`—The trigger is disconnected and can't receive events due to lack of permissions. • `Expired`—The trigger's connection expired. In rare cases, subscriptions can expire if they're inactive for an extended period of time.
`Tip`	**Type** int **Properties** Filter, Group, Nillable, Sort **Description** The replay ID of the last published event.
`Topic`	**Type** string **Properties** Filter, Group, Nillable, Sort **Description** 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.
`Type`	**Type** string **Properties** Filter, Group, Nillable, Sort

Field	Details
	Description The subscriber type (`ApexTrigger`). If the subscriber is a process or flow Wait element, the type is blank.

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.

```
SELECT ExternalId, Name, Position, Status, Tip, Type
FROM EventBusSubscriber
WHERE Topic='Low_Ink__e'
```

EventBus Class

Contains methods for publishing platform events.

Namespace

System

EventBus Methods

SEE ALSO:

Platform Events Developer Guide: Publishing Platform Events

EventBus Methods

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

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. Otherwise, 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.

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

```
public static List<Database.SaveResult> 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.SaveResult>

A list of results, each corresponding to the result of publishing one event. 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. Otherwise, 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.

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.

TriggerContext Class

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

Namespace

EventBus

TriggerContext Properties

The following are properties for `TriggerContext`.

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

```
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.

```
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

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

Property Value

Type: Integer

TriggerContext Methods

The following are methods for `TriggerContext`.

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

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

Return Value

Type: EventBus.TriggerContext

Information about the currently executing trigger.