Platform Events (Beta)

Salesforce, Winter '17

@salesforcedocs

Last updated: December 9, 2016

CONTENTS

PLATFORM EVENTS (BETA)

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.

EDITIONS

Platform Events is available in both Lightning Experience and Salesforce Classic. The definition of platform events is available in Salesforce Classic only.

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

USER PERMISSIONS

To create and edit platform event definitions:
- "Customize Application"

Note: This release contains a beta version of Platform Events, which means it's a high-quality feature with known limitations. For information on enabling this feature in your org, contact Salesforce. Platform Events isn't generally available unless or until Salesforce announces its general availability in documentation or in press releases or public statements. We can't guarantee general availability within any particular time frame or at all. Make your purchase decisions only on the basis of generally available products and features. You can provide feedback and suggestions for Platform Events in the Success Community.

By using platform events, publishers can send customized event data through Apex or an API. Subscribers can receive custom notifications from Salesforce or an external system and respond with actions using Apex or CometD clients. For example, a printer can make an API call to publish an event when the ink is low. The custom printer event can contain custom fields for the printer model, serial number, and ink level. The event is processed in Salesforce by an Apex trigger that places an order for a new cartridge.

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. Multiple subscribers can listen to the same event and carry out different actions.

Define Your Platform Event

Define your platform event in the Platform Events page in the Salesforce user interface. From Setup, enter *`Platform Events`* in the `Quick Find` box, then select **Platform Events**.

Platform events are sObjects, similar to custom objects but with some limitations. Event notifications are instances of platform events. Unlike sObjects, you can't update event notifications. You also can't view them in the user interface. When you delete platform event definitions, they're permanently deleted.

Publish and Subscribe to Platform Events

After you define your platform event, you can publish event notifications and subscribe to events using Apex or an API.

In Apex, you publish event notifications by inserting event records with the `EventBus.publish` static method. To receive published notifications, write an `after insert` trigger on your event object. In the trigger, you can inspect each event notification and perform some business logic. You don't need to create a channel, because Salesforce creates a channel for each defined platform event.

Using an API, you 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. Unlike Apex, there is no extra call to publish the event. Subscribe to events with CometD clients or by writing Apex `after insert` triggers on your event object.

Standard Fields

The following standard fields appear on the New Platform Event page and are included in all platform events. In addition to these standard fields, your event can include custom fields that you define.

Field	Description
`Label`	Name used to refer to your platform event in a user interface page.
`Plural Label`	Plural name of the platform event.
`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 custom events when you are viewing them in a list.
`Deployment Status`	Indicates whether the platform event is visible to other users.

`ReplayId` System Field

Among the standard fields that the system populates for each custom platform event is a number field called `ReplayId` that identifies an event. Each replay ID is guaranteed to be higher than the ID of the previous event, but not necessarily contiguous for consecutive events. The ID is unique for the org and the channel, and is used to replay past events.

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.

Event Subscribers

The Subscriptions related list shows all the triggers that are subscribed to platform events. The related list shows the last replay ID of the platform event that each subscription has processed and whether errors occurred. Also, information about event subscribers is exposed in the EventBusSubscriber object. You can query this object to obtain details about subscribers.

Platform Event Considerations

Permanent Deletion of Event Definitions

When you delete an event definition, it is permanently deleted and can't be restored. Ensure that you delete any associated triggers first before you delete the event definition. Any published events whose definition has been deleted are also permanently deleted.

Renaming Event Objects

Before you rename an event, ensure that you delete any 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, re-add your trigger for the renamed event object.

No Associated Tab

Platform events have no associated tab because you can't view event records in the Salesforce user interface.

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.

Limit for Published Events

You can publish up to 10,000 platform events per hour.

Custom Field Attributes

The Required and Default Value field attributes aren't enforced for platform event custom fields. Also, other attributes aren't enforced, such as the precision of number fields and the maximum length of text fields.

Supported Field Types

Platform event custom fields support only these field types: Text, Text Area (Long), Number, Date, Date/Time, and Checkbox.

Replaying Past Events

You can replay platform events that were sent in the past 24 hours. You can replay platform events through the API but not Apex. The process of replaying platform events is the same as for other Streaming API events. For more information, see the Replay Code Example in the Streaming API Developer Guide and Streaming Replay Client Extensions for Java and JavaScript on GitHub.

SEE ALSO:

EventBusSubscriber

PLATFORM EVENTS IN APEX

Publish platform events in Apex by calling the `EventBus.publish` method, and subscribe to notifications through Apex triggers.

Publish Platform Event Notifications

To publish event notifications, 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.

The following example creates two events of type `Low_Ink__e`, publishes them, and then checks if the publishing was successful or any errors were encountered. The example requires the `Low Ink` platform event to be 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());
        }
    }
}
```

When you publish events from Apex, they're inserted synchronously. Because event publishing is equivalent to a DML insert operation, DML limits apply.

Subscribe to Platform Event Notifications with Triggers

Use triggers to subscribe to events. 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 desired event object type. In this context, the `after insert` trigger event corresponds to the time that occurs after a platform event is published. After an event notification is published, the `after insert` trigger for this event object is fired.

The following example shows a trigger for the `Low Ink` event. It iterates through each event and checks a field value. This trigger inspects each received notification and gets the printer model from the notification. If the printer model matches a certain value, other business logic can be executed. For example, the trigger can create a request to order a new cartridge for this printer model.

```
trigger LowInkTrigger on Low_Ink__e (after insert) {
    // Trigger for catching Low_Ink events.
    // Iterate through each notification.

    // List to hold all cases to be created.
    List<Case> cases = new List<Case>();
    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;
            cases.add(cs);
        }
    }

    // Insert all cases corresponding to events received.
    insert cases;
}
```

Refire Event Triggers with Resent Events

The ability to refire event triggers gives you another chance to process event notifications. Refiring triggers is helpful when a transient error occurs or when waiting for a condition to change. Refire triggers if the error or condition is external to the event records and is likely to go away later. For example, consider that the trigger is trying to add a related record to a master record only if a field on the master record equals a certain value. It is possible that in a subsequent try, this field value changes and the trigger is able to perform the operation.

To refire the event trigger, throw `EventBus.RetryableException`. The system resends events after a small delay, which increases on each subsequent retry. A resent event has the same field values as the original event, but the batch size of the events can be different. 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.

This example is a skeletal trigger that gives you an idea of how to throw `EventBus.RetryableException`. This 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 {
        // Condition isn't met, so try again later.
        throw new EventBus.RetryableException();
    }
}
```

Copy Event Field Values with `ObjectMapper`

The Apex `ObjectMapper` class provides utility methods for copying field values from events to instances of a Salesforce object (standard or custom object). Use `transform` methods to shorten the amount of code you have to write to copy field values. The `ObjectMapper` class provides several methods named `transform` that take different parameters.

This example illustrates how to use an `ObjectMapper.transform` method to copy field values within a trigger. The event records, which correspond to event notifications, are supplied in the `Trigger.New` trigger variable. The sObjects that receive the copied values correspond to a custom object called `MyCustomObject__c`. The result of the copy is stored in an array of sObjects.

```
SObject[] results = Databinding.ObjectMapper.transform('MyCustomObject__c', Trigger.new);
```

To obtain an array of your custom object instead of the generic sObjects, cast the returned array to your object, as follows.

```
MyCustomObject__c[] results =
(MyCustomObject__c[])Databinding.ObjectMapper.transform('MyCustomObject__c', Trigger.new);
```

This example shows the amount of code you would have to write to copy event field values when not using `ObjectMapper`. The `ObjectMapper` class methods save you a lot of time!

```
List<MyCustomObject__c> results = new List<MyCustomObject__c>();
// Copy field values from events without the ObjectMapper class.
for (Low_Ink__e e : Trigger.new) {
    MyCustomObject__c c = new MyCustomObject__c();
    c.Printer_Model__c = e.Printer_Model__c;
    c.Serial_Number__c = e.Serial_Number__c;
    c.Ink_Percentage__c = e.Ink_Percentage__c;
    results.add(c);
}
```

If the names of the fields in the sObject don't match the event field names, use the `transform` method that takes an additional Map parameter. The passed-in map converts event field names to the sObject field names.

```
Map<String,String> map = new Map<String,String>();
map.put('FieldName1OnEvent__c', 'FieldName1OnCustomObject__c');
map.put('FieldName2OnEvent__c', 'FieldName2OnCustomObject__c');
SObject[] results = Databinding.ObjectMapper.transform('MyCustomObject__c', Trigger.new,
map);
```

Note: The `transform` methods automatically map between standard fields and custom fields with the same name. For example, if you copy field values from an event record to an account, values in an AccountNumber__c custom field on the event are mapped to an AccountNumber standard field, if it exists.

To learn about the available methods, see the `ObjectMapper` class.

Apex and SOQL Considerations for Platform Events

Trigger Support for `after insert` Triggers Only

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 daily event 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.

No SOQL Support

You can't query event notifications using SOQL.

Apex DML Limits for Publishing Events

Each `EventBus.publish` method call is considered a DML statement, and DML limits apply.

SEE ALSO:

EventBus Class

PLATFORM EVENTS IN THE API

Publish event notifications through the API, and subscribe with a Streaming API client that uses CometD or through Apex triggers. Using the API, you can publish notifications from external systems.

Publish Platform Event Notifications with Salesforce APIs

Publishing a platform event consists of creating the sObject record, the same way you create records for any other sObject. You can use any Salesforce API that creates platform events, such as SOAP API, REST API, or Bulk API. Unlike Apex, there is no extra call to publish events.

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/v37.0/sobjects/Low_Ink__e/
```

Request body:

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

After the platform event record is created, the REST response looks as follows. Headers are deleted for brevity.

```
HTTP/1.1 201 Created

{
   "id" : "e00xx000000000B",
   "success" : true,
   "errors" : [ ]
}
```

For more information, see the Force.com REST API Developer Guide.

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>HP_Envy-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>Cannon-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>
```

For more information about creating records with the SOAP API, see the `create()` call in the SOAP API Developer Guide.

Subscribe to Platform Event Notifications with CometD

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 format of the platform event channel name is as follows.

```
/event/<EventName>__e
```

For example, if you've defined a platform event named `Low Ink`, provide this channel name when subscribing.

```
/event/Low_Ink__e
```

Also ensure that your API client uses version 37.0 of the CometD endpoint.

```
/cometd/37.0
```

To learn how to add a Java API client that uses CometD, check out the Java Client example in the Streaming API Developer Guide. Modify the following constants in the example to provide suitable values for your org. The given channel name value is based on our example event named `Low Ink`.

Constant	Value
CHANNEL	`/event/Low_Ink__e`
STREAMING_ENDPOINT_URI	`/cometd/37.0`
USER_NAME	A valid username for your org
PASSWORD	Your user's password

Next, 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.

Subscribe to Platform Event Notifications with Apex Triggers

Instead of using a CometD client, you can subscribe to events published through the API by using Apex triggers. Inserting the platform event record through the API fires the associated Apex `after insert` trigger. For more information, see Subscribe to Platform Event Notifications with Triggers.

API and SOQL Considerations for Platform Events

No SOQL Support

You can't query event notifications using SOQL.

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.

EVENTBUSSUBSCRIBER

Represents a trigger 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.
`Position`	**Type** int **Properties** Filter, Group, Nillable, Sort **Description** The replay ID of the last event that the subscriber processed.
`Status`	**Type** picklist

Field	Details
	Properties Filter, Group, Nillable, Restricted picklist, Sort **Description** Indicates the status of the subscriber. Can be one of the following values: • `Running`—The subscriber is actively listening to events. • `Suspended`—The subscriber is disconnected and can't receive events due to lack of permissions. • `Expired`—The subscriber's connection expired. In rare cases, subscriptions can expire if they're inactive for an extended period of time. • `Error`—The subscription encountered an error and has been disconnected.
`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 **Description** The subscriber type. Can be one of the following values: • `ApexTrigger` • `Process`—Reserved for future use.

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

Usage

To learn how to use platform events in Apex, see Platform Events in Apex.

EventBus Methods

EventBus Methods

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

publish(event)
Publishes the given platform event. To receive published events, use triggers for the corresponding event object.

publish(events)
Publishes the given list of platform events. To receive published events, use triggers for the corresponding event object.

`publish(event)`

Publishes the given platform event. To receive published events, use triggers for the corresponding event object.

Signature

```
public static Database.SaveResult publish(SObject event)
```

Parameters

`event`
Type: SObject

An instance of a platform event. You must define your platform event object first in your org. For example, the type of the platform event object can be `MyEvent__e`.

Return Value

Type: Database.SaveResult

The result of publishing the given event.

Usage

Note: This method inserts events synchronously. The insertion is part of an Apex transaction. Apex DML limits, such as number of records processed in DML statements, apply to this method.

`publish(events)`

Publishes the given list of platform events. To receive published events, use triggers for the corresponding event object.

Signature

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

Parameters

`events`
Type: List<sObject>

  A list of platform event instances. You must define your platform event object first in your org. For example, the type of the platform event object can be `MyEvent__e`.

Return Value

Type: List<Database.SaveResult>

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

Usage

Note: This method inserts events synchronously. The insertion is part of an Apex transaction. Apex DML limits, such as number of records processed in DML statements, apply to this method.

OBJECTMAPPER CLASS

Provides utility methods for copying field values from platform events to Salesforce objects.

Namespace

Databinding

ObjectMapper Methods

ObjectMapper Methods

The following are methods for `ObjectMapper`.

transform(targetType, sourceRecord)
: Copies the field values from the specified platform event record to an instance of a Salesforce object.

transform(targetType, sourceRecords)
: Copies the field values from the specified platform event records to instances of Salesforce objects.

transform(targetType, sourceRecord, fieldMapping)
: Copies the field values from the specified platform event record to an instance of a Salesforce object. Use this method when field names don't match between the event and the Salesforce object.

transform(targetType, sourceRecords, fieldMapping)
: Copies the field values from the specified platform event records to instances of a Salesforce object. Use this method when field names don't match between the event and the Salesforce object.

`transform(targetType, sourceRecord)`

Copies the field values from the specified platform event record to an instance of a Salesforce object.

Signature

```
public static SObject transform(String targetType, SObject sourceRecord)
```

Parameters

`targetType`

Type: String

The type of the returned sObject that contains the copied field values. This parameter can be a standard object or a custom object. For example, MyCustomObject__c or Account.

sourceRecord

Type: SObject

A platform event record containing the fields to copy.

Return Value

Type: SObject

An instance of the specified standard or custom object containing the copied fields.

Example

In this example, the `transform` method returns an sObject containing the field values from the specified event record. The type of the sObject created is MyCustomObject__c, which the result can be cast to.

```
SObject result = Databinding.ObjectMapper.transform('MyCustomObject__c', eventRecord);
```

transform(targetType, sourceRecords)

Copies the field values from the specified platform event records to instances of Salesforce objects.

Signature

```
public static List<SObject> transform(String targetType, List<SObject> sourceRecords)
```

Parameters

targetType

Type: String

The type of the returned sObject that contains the copied field values. This parameter can be a standard object or a custom object. For example, MyCustomObject__c or Account.

sourceRecords

Type: List<SObject>

A list of platform event records containing the fields to copy.

Return Value

Type: List<SObject>

Instances of the specified standard or custom object containing the copied fields for each platform event.

Example

In this example, the `transform` method returns an array of sObjects containing the field values from the event records specified in `Trigger.new`. The type of the sObjects created is MyCustomObject__c, which the result objects can be cast to.

```
SObject[] results = Databinding.ObjectMapper.transform('MyCustomObject__c', Trigger.new);
```

`transform(targetType, sourceRecord, fieldMapping)`

Copies the field values from the specified platform event record to an instance of a Salesforce object. Use this method when field names don't match between the event and the Salesforce object.

Signature

```
public static SObject transform(String targetType, SObject sourceRecord,
Map<String,String> fieldMapping)
```

Parameters

targetType
Type: String

The type of the returned sObject that contains the copied field values. This parameter can be a standard object or a custom object. For example, MyCustomObject__c or Account.

sourceRecord
Type: SObject

A platform event record containing the fields to copy.

fieldMapping
Type: Map<String,String>

A map of event field names to your object's field names. This map determines in which field to copy each event field value.

Return Value

Type: SObject

An instance of the specified standard or custom object containing the copied fields.

Example

In this example, a map is created that maps two event field names to corresponding field names on the sObject. This map is passed in to the `transform` method as the last parameter.

```
Map<String,String> map = new Map<String,String>();
map.put('FieldName1OnEvent__c', 'FieldName1OnCustomObject__c');
map.put('FieldName2OnEvent__c', 'FieldName2OnCustomObject__c');
SObject result = Databinding.ObjectMapper.transform('MyCustomObject__c', eventRecord, map);
```

`transform(targetType, sourceRecords, fieldMapping)`

Copies the field values from the specified platform event records to instances of a Salesforce object. Use this method when field names don't match between the event and the Salesforce object.

Signature

```
public static List<SObject> transform(String targetType, List<SObject> sourceRecords,
Map<String,String> fieldMapping)
```

Parameters

targetType

Type: String

The type of the returned sObject that contains the copied field values. This parameter can be a standard object or a custom object. For example, MyCustomObject__c or Account.

sourceRecords

Type: List<SObject>

A list of platform event records containing the fields to copy.

fieldMapping

Type: Map<String,String>

A map of event field names to your object’s field names. This map determines in which field to copy each event field value.

Return Value

Type: List<SObject>

Instances of the specified standard or custom object containing the copied fields for each platform event.

Example

In this example, a map is created that maps two event field names to corresponding field names on the sObject. This map is passed in to the `transform` method as the last parameter.

```
Map<String,String> map = new Map<String,String>();
map.put('FieldName1OnEvent__c', 'FieldName1OnCustomObject__c');
map.put('FieldName2OnEvent__c', 'FieldName2OnCustomObject__c');
SObject[] results = Databinding.ObjectMapper.transform('MyCustomObject__c', eventRecords,
 map);
```

DATABINDING EXCEPTION

The `Databinding` namespace contains an exception class.

Exception	Description
`Databinding.ObjectMappingException`	This exception is thrown when the `transform` method encounters an error while copying field values. Note: This exception isn't thrown when the fields or field types between the event and the custom object don't match. Instead, the values of the matching fields get copied and the mismatched fields are ignored.