OmniCloud Pro Utilities

Developer Guide

OmniCloud Pro Utilities is a Salesforce managed package that provides utility classes to assist developers in implementing custom solutions using best practices. The purpose of this managed package is to provide a consistent means for implementing triggers.

Initial Setup

To grant access to configuring OmniCloud Pro Utilities, add the “OmniCloud Pro Utilities Administrator” permission set to the users that should have access.

Trigger Factory Utility Classes

Implementing a Trigger Handler

OmniCloud Pro Utilities provides the necessary classes so that triggers can be developed using a Factory pattern. Each trigger implemented must be accompanied by a trigger handler class, which the trigger invokes.

For example, an account trigger that is invoked on all seven events (before insert, before update, before delete, after insert, after update, after delete, after undelete) would look like this:

```
trigger MyAccountTrigger on Account (before insert, before update, before delete,
        after insert, after update, after delete, after undelete) {
     OmniCloudPro.TriggerFactory.invoke('MyAccountHandler');
}
```

The parameter in the `OmniCloudPro.TriggerFactory.invoke(String)` method is the name of the trigger handler class that either implements the `OmniCloudPro.Triggerable` interface or extends `OmniCloudPro.TriggerableAdapter` class. If the trigger handler class extends `OmniCloudPro.TriggerableAdapter`, then it does not need to implement all of the methods in `OmniCloudPro.Triggerable` since it already implements that interface. The trigger handler class for the aforementioned trigger could look like this:

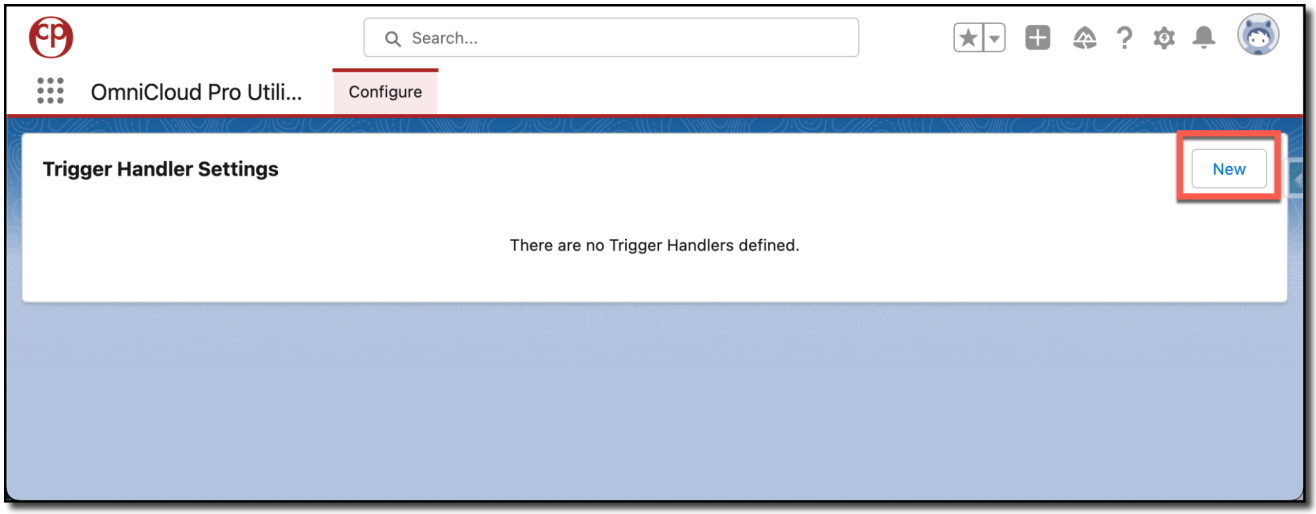
```
global without sharing MyAccountHandler extends OmniCloudPro.TriggerableAdapter {

    global override void oncePreBefore() {
        // Logic that executes once at the beginning of any before-phase event.
    }

    global override void eachBeforeInsert(SObject newRecord) {
        // Logic that executes for each record in the "before insert" event.
    }

    global override void eachBeforeUpdate(SObject oldRecord, SObject newRecord) {
```

```
        // Logic that executes for each record in the "before update" event.
    }

    global override void eachBeforeDelete(SObject oldRecord) {
        // Logic that executes for each record in the "before delete" event.
    }
    global override void oncePostBefore() {
        // Logic that executes once at the end of any before-phase event.
    }

    global override void oncePreAfter() {
        // Logic that executes once at the beginning of any after-phase event.
    }

    global override void eachAfterInsert(SObject newRecord) {
        // Logic that executes for each record in the "after insert" event.
    }

    global override void eachAfterUpdate(SObject oldRecord, SObject newRecord) {
        // Logic that executes for each record in the "after update" event.
    }

    global override void eachAfterDelete(SObject oldRecord) {
        // Logic that executes for each record in the "after delete" event.
    }

    global override void eachAfterUndelete(SObject newRecord) {
        // Logic that executes for each record in the "after undelete" event.
    }

    global override void oncePostAfter() {
        // Logic that executes once at the end of any after-phase event.
    }

}
```

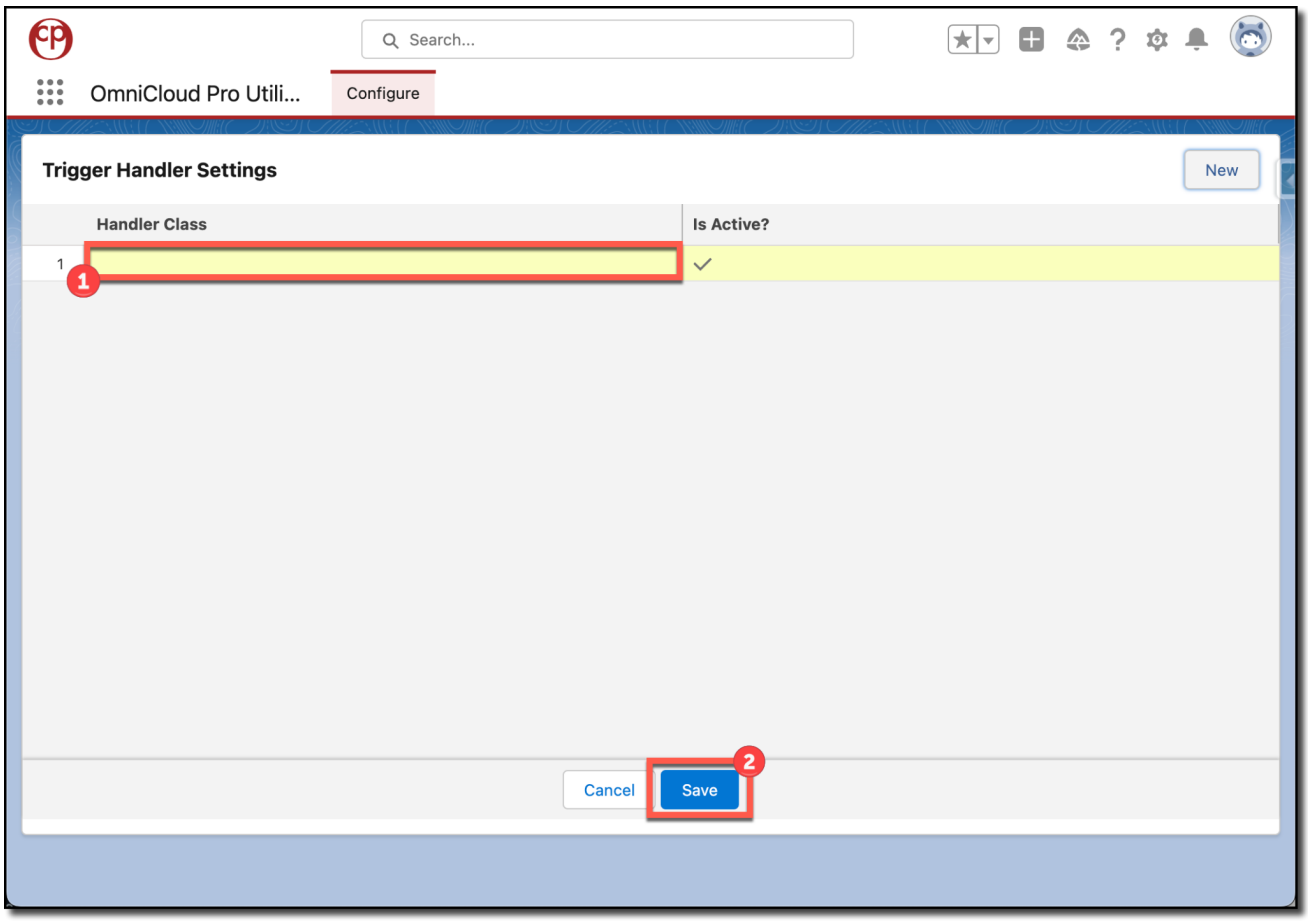
The trigger handler class must be declared as `global without sharing`. It needs to be declared as `global` so that the `OmniCloud.TriggerFactory` class can access it. It needs to be declared as `without sharing` to prevent issues from invocations executed by users with more restrictive Salesforce license types.

Once both the trigger and the handler class are implemented, it needs to be added and activated on the “Configure” tab in the “OmniCloud Pro Utilities” app.

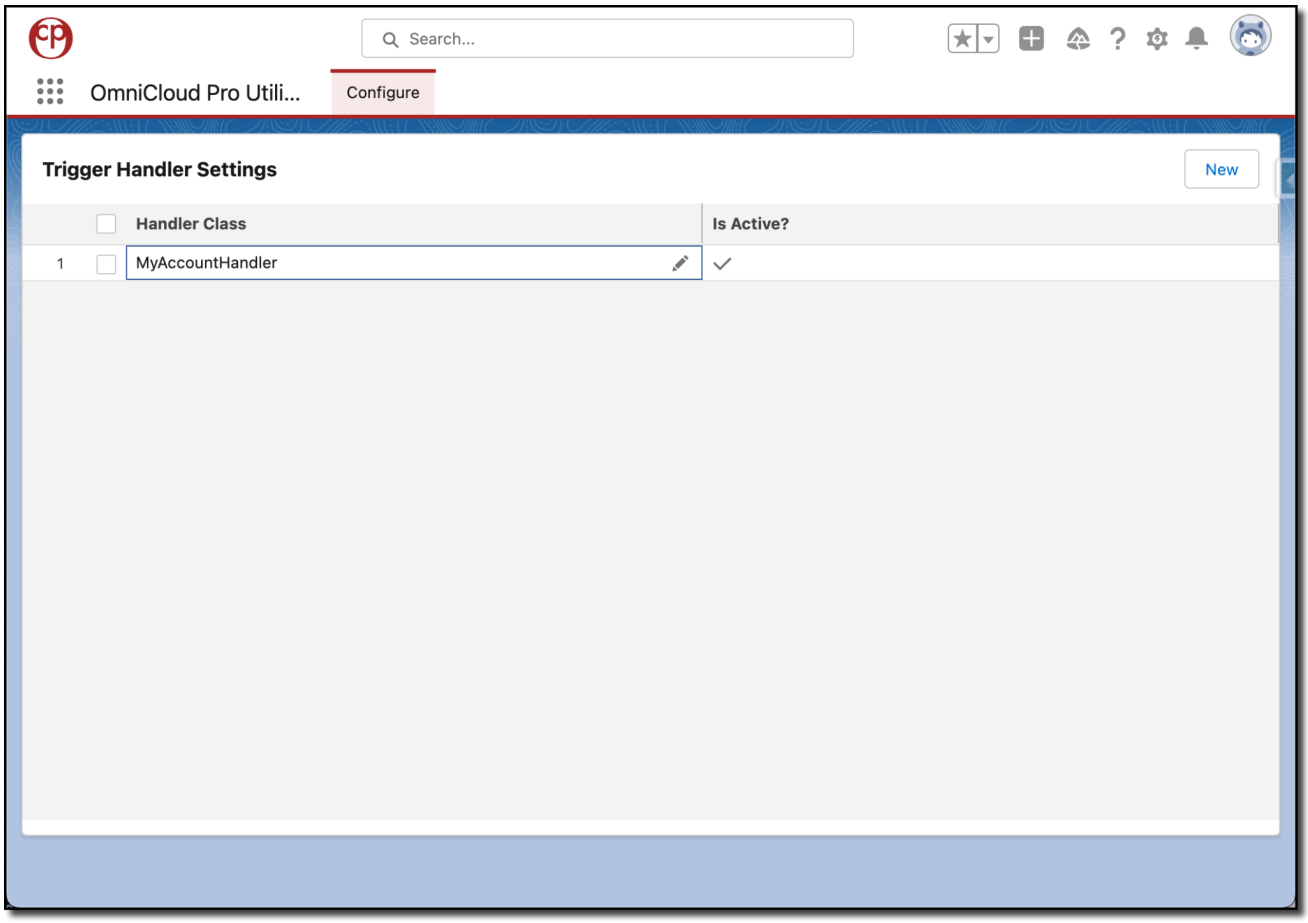
1. On the “Configure” tab, click the “New” button:



2. Enter the name of the Handler Class in the highlighted field, then click the “Save” button:



3. Once saved, it should look like this:



Trigger Implementation Tips

- Use `oncePreBefore()` and `oncePreAfter()` methods to cache records and perform data setup.
- Implement assignment and validation logic in the `eachBefore*()` methods.
- Use the `oncePostBefore()` method to perform summary calculations.
- Use the `oncePostAfter()` method to perform DML.

Mitigating Trigger Recursion

As the complexity and customization of a Salesforce organization tend to grow over time, there is an increased likelihood that one trigger may invoke another trigger (or itself multiple times), which can result in reaching Apex governor limits. The `OmniCloudPro.TriggerRunStatus` and `OmniCloudPro.TriggerRunStatusItem` classes can be used to mitigate this risk. For example, if you want to ensure DML statements in the `oncePostAfter()` method are executed once, the `oncePostAfter()` methods would be implemented as follows:

```
global without sharing class MyOpportunityHandler
        extends OmniCloudPro.TriggerableAdapter {

    private static final String TRIGGER_RUN_STATUS_ID = 'MyOpportunityHandler';

    private Map<Id, Account> accountRecordsToUpdateMap = new Map<Id, Account>();
```

```
    // ... Other trigger handler methods ...

    global override void oncePostAfter() {

        // Prevent trigger recursion
        if (!OmniCloudPro.TriggerRunStatus.isRunnable(
                new OmniCloudPro.TriggerRunStatusItem(TRIGGER_RUN_STATUS_ID))) {
            return;
        }
        OmniCloudPro.TriggerRunStatus.block(
                new OmniCloudPro.TriggerRunStatusItem(TRIGGER_RUN_STATUS_ID));

        // Notify account records to update
        if (!accountRecordsToUpdateMap.isEmpty()) {
            update accountRecordsToUpdateMap.values();
            accountRecordsToUpdateMap.clear();
        }

    }

}
```

In the trigger handler class above, the constant, `TRIGGER_RUN_STATUS_ID`, stores the name of the trigger handler, which is used as the parameter to the `OmniCloudPro.TriggerRunStatusItem` constructor. The `OmniCloudPro.TriggerRunStatusItem` class keeps track of the record IDs that are in the trigger context and assigns it to the provided name (in this case, `TRIGGER_RUN_STATUS_ID`). Invoking `OmniCloudPro.TriggerRunStatus.block()` with the `OmniCloudPro.TriggerRunStatusItem` object, flags that this trigger handler has already run for the records in the trigger context. Invoking `OmniCloudPro.TriggerRunStatus.isRunnable()` with the `OmniCloudPro.TriggerRunStatusItem` object returns `true` if the following trigger logic should not be blocked from execution, and `false` otherwise. The state of the `OmniCloudPro.TriggerRunStatus` class persists for the life of the transaction.

> The `OmniCloudPro.TriggerRunStatusItem` class cannot track the records during the `before insert` phase. This is because the IDs for the records in the trigger context are not established until the `after insert` phase.

Testing Trigger Handlers

OmniCloud Pro Utilities facilitates the unit testing of triggers, especially if the test class is isolating test data from organization data (that is, not setting `IsAllData=true` in the `@IsTest` annotation). In this case, when the unit test runs, no trigger handlers are active since no settings exist.

There are two ways to activate trigger handlers in unit test:

- To activate a single handler, use the `OmniCloudPro.TriggerHandlerSettings.activate(String)` method. For example:

```
OmniCloudPro.TriggerHandlerSettings.activate('MyAccountHandler');
```

- To activate multiple handlers use the `OmniCloudPro.TriggerHandlerSettings.activate(Set<String>)` method. For example:

```
OmniCloudPro.TriggerHandlerSettings.activate(new Set<String> {
        'MyAccountHandler',
        'MyOpportunityHandler'
});
```

This allows you to test specific trigger handlers in a unit test.

API Documentation

OmniCloud Pro API documentation is located on https://apidocs.omnicloudpro.com/OmniCloudProUtilities.