



SALESFORCE INTEGRATION

A GENERIC APPROACH TO CALL
REST SERVICES FROM SALESFORCE

Jack van Dijk

Salesforce

Introduction	3
Administration screens	3
Integration Target	4
Integration Definition	4
Integration Input	4
<i>Body Element</i>	4
<i>URI Argument</i>	4
<i>Header Variable</i>	4
<i>Query Parameter</i>	5
Integration Output	5
Generic Apex Classes	5
RESTCallout	6
RESTDataSourceConnection	7
Supporting Interfaces	7
RESTMockService	7
RESTDataConversion	7
Using the RESTCallOut class	8
Triggering the call from Process Builder	8
Using the RESTCallOut class from Apex	9
Alternative – Platform Events	10
Improvement – Platform Cache	10

Introduction

This document describes the method implemented to reduce the amount of Apex code needed to perform callouts from Salesforce to an external system while using REST calls and to maximize re-use.

Most REST service calls show a similar pattern and can roughly be described as follows:

- Collect some data that needs to be sent as part of the service call
- Perform the actual REST call
- Process the result that the service returns.

The second scenario is that the service will return a list of records of a certain type. This is typical a good scenario for using External Objects, but again we want to make this as flexible as possible.

While we can't do this without writing Apex at all, we want to minimize duplication and reduce the lines of code.

Administration screens

In order to define the details for an integration in a flexible way we've created some new objects and UI with that to manage this process.

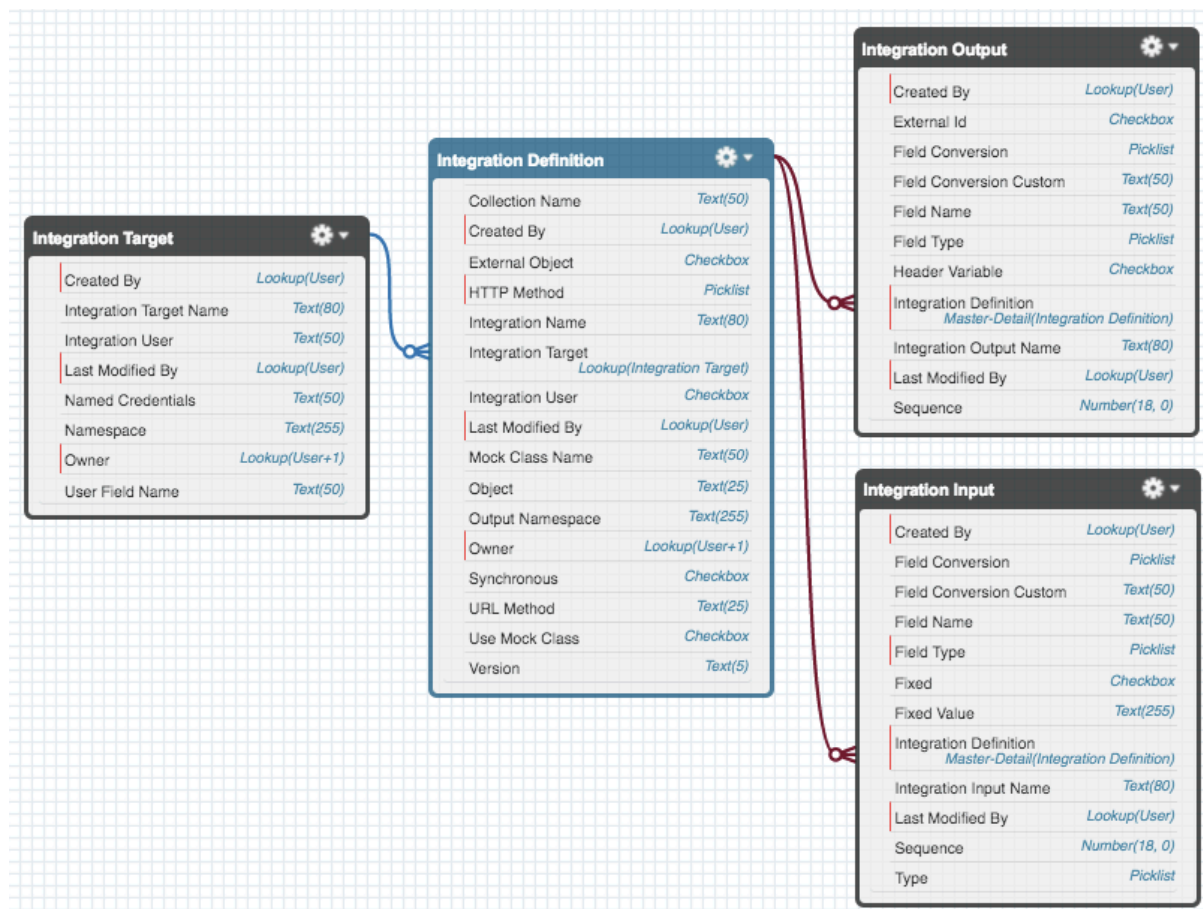


Figure 1 Integration objects schema

In the above diagram, we have 4 objects that we use for the administration of the callouts.

Integration Target

The Integration Target is the object where we store generic information about the target system where we need to integrate with. Things like the integration user and the Named Credential to use are stored here.

Integration Definition

The Integration Definition is the main object for a specific call. It will hold details about the REST url method and the Salesforce object that is used. Also, we can specify if this is a call for a single record or if we are getting a list of records back (External Object)

Integration Input

Integration Input list the input arguments for the REST call. An input argument can be either a constant or a field that we will retrieve from the object that we have specified on the Integration Definition.

There can be 4 different types of input parameters defined: Body Element, Header Variable, Query Parameter or an argument in the URI.

Body Element

This is the most common type of input parameter. The parameter will be send as part of the JSON structure in the body of the call. Based on the dot notation the parameter can be made part of the of the top level or as a child object in the JSON. For example when it is called LastName it will appear in the JSON like:

```
{
  "LastName": "<some value>"
}
```

But when we name the parameter customer.LastName the JSON will become like this:

```
{
  "customer": {
    "LastName": "<some value>"
  }
}
```

URI Argument

An URI argument is a variable that is made part of the URL that we are calling. This is typically used to select a certain record. For example the following url with the URI Argument in bold:

```
/services/data/v37.0/objects/Account/0019E000002pDAG
```

Header Variable

A header variable is stored as a name/value pair in the HTTP header of the request. This is typically used for user, process or session information.

Query Parameter


A query parameter is a way to append extra parameters to the url, for example:

`/Account/0019E0002pDAG?Argument1=Value1&Argument2=Value2`

In this example Argument1 and Argument2 are two query parameters being used.

Integration Output

Below is an example of a Blacklist call to an external system in the administration screen:


 INTEGRATION DEFINITION

Blacklist

Object	Integration Target	URL Method
Contact	Pega	blacklistcheck

RELATED


DETAILS

 Integration Inputs (3)

New

INTEGRATION INPUT NAME	FIXED	FIXED VALUE	SEQUENCE
customer.firstName	<input type="checkbox"/>		1
customer.lastName	<input type="checkbox"/>		2
customer.ssn	<input checked="" type="checkbox"/>	123456789	3

View All

 Integration Outputs (1)

New

INTEGRATION OUTPUT NAME	FIELD NAME	FIELD TYPE
result	Compliant__c	Boolean

View All

Figure 2 Integration Administration page

Generic Apex Classes

There are two different classes being used:

- **RESTCallout**, this is the class that supports single record calls to a rest service.
- **RESTDataSourceConnection**, this is the generic custom datasource of which the External Objects can be generated.

RESTCallout

The generic Callout class is an Apex class that reads the data from the Administration screens and then creates the input JSON for the REST call, performs the actual REST call and then processes any return values from the call.

The process has been divided in a number of steps, this gives us a flexible way to use the functionality. In general, you'll see the following pattern:

1. Get the call definitions (LoadIntegrationStructure)
2. Link the records to the definition (SetIds)
3. Load the data from the records needed (LoadObjectData)
4. Perform the REST call (PerformRESTCalls)
5. Process the return values of the REST call (SaveObjectData)

By splitting this in three different function we've got the flexibility to group multiple REST calls when needed. This also allows us to keep data in memory when performing multiple calls in a row which gives us additional flexibility.

In this class there are a number of functions defined:

- **ProcessBuilderRESTCall**, this is the method that we call from a process flow. This method is passed a single parameter from the flow. Because the Salesforce platform will automatically bulkify calls when possible the method needs to assume that a array of input parameters is passed in. It will loop through those inputs and call the REST method for each of them.
- **PerformRESTCallAsync**, this performs an Async REST call for all records of a given Integration Definition. This method is an asynchronous wrapper around the next function PerformRESTCall.
- **PerformRESTCall**, this method loops for a set of records and calls the REST method for each of them.
- **PerformSingleRESTCall**, this is the method that performs the actual REST call.
- **PerformRESTCalls**, this is the overall function that decides to do the calls synchronous or asynchronous.
- **LoadObjectData**, this uses the query that was determined to get the data from the Salesforce object needed to do the REST call.
- **SaveObjectData**, this saves the results from the REST calls to the Salesforce objects in the database.
- **LoadIntegrationStructure**, this loads the integration definition for the given methods from database. These are the definitions, including input and output arguments that are defined in the administrative screen.
- **SetIds**, this associates Ids of Salesforce objects with a given REST call.
- **CreateObjectQuery**, based on the object specified on the integration definition and the fields specified on the input arguments a query is constructed to retrieve data from the database.
- **FinalizeObjectQueries**, the query constructed in the previous call is completed with the IDs that are set using the SetIds function.
- **GetFieldValue**, this method translates data types like Boolean and Dates from the Salesforce format into the REST format.

- **SetFieldValue**, this method translates data types like Boolean and Dates from the REST format into the Salesforce format.

RESTDataSourceConnection

The RESTDataSourceConnection is the generic custom Salesforce Connect datasource. It has two main methods:

- **Sync**, this function is called by the Salesforce config when you generate an External Object. Here we create the tables and columns that we need based on the definitions created in the Admin screen.
- **Query**, this method needs to retrieve the data to fill the table and columns as defined in the List method and performs the REST call to the external system to get the data.

The RESTDataSourceConnection re-uses a lot of the functionality that is already part of the RESTCallout class.

Supporting Interfaces

Next to the core Apex classes there have be two interfaces defined to be able to extend the framework.

RESTMockService

The RESTMockService interface allows you to implement mock classes that return mock results for one of your integration. On integration level in the administration screens you can specify the name of your mock class that it should use. There is also a checkbox 'Use Mock class' that determines if it will call the mock class for the result or perform the actual REST call.

The RESTMockService has one single method that needs to be implemented in your mock class called GetMockResponse which returns a HTTPResponse object with needs to have your result values.

RESTDataConversion

The RESTDataConversion interface allows you to write data conversion between data coming back from the REST call and the how Salesforce expects it and the other way around. In the administration screens for an Input and Output you can already specify for some typical use-cases the data conversion to apply. These conversions that are pre-packaged are also based on this interface. Other conversion can be created and then the class name of the conversion class can then specified on an input or output when the Custom conversion method is chosen.

Standard conversions that are already delivered are:

- Conversion between Salesforce Boolean and Yes/No values.
- Conversion between Salesforce Boolean and True/False values.
- Conversion between Salesforce Date object and string in DD/MM/YYYY format

- Conversion between Salesforce Date object and string in MM/DD/YYYY format
- Conversion between Salesforce Date object and a Julian Date number.

The interface has two methods that need to be implemented when writing your own custom conversion:

1. ExternalFormatToSalesforce, and
2. SalesforceToExternalFormat

The interface

Using the RESTCallOut class

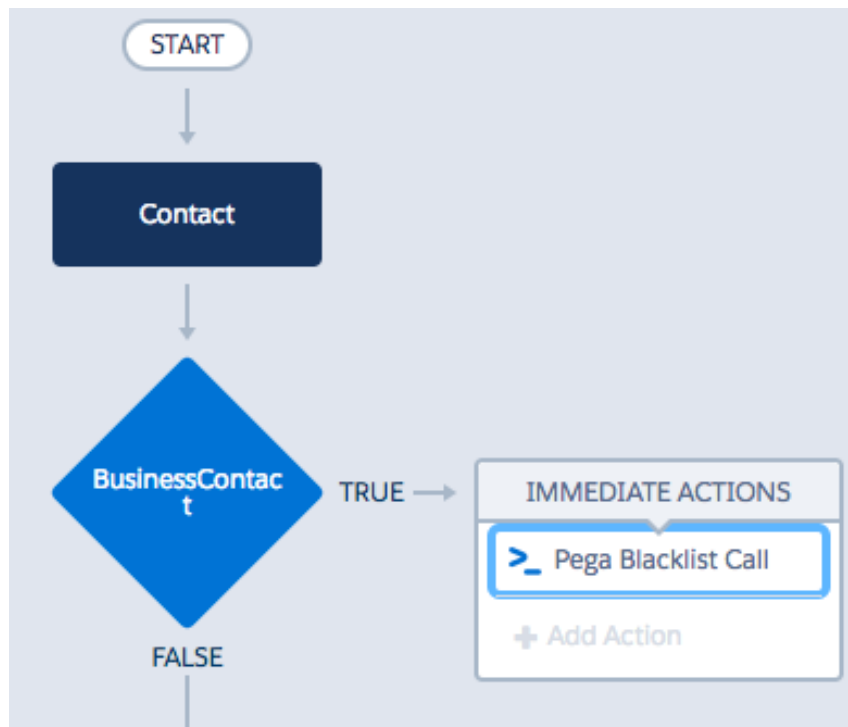
There are two different scenarios in which the RESTCallOut class is being used. One is while calling it from the Process Builder. These are typically the calls around a single record where data needs to be sent to the external system and a response is received which is stored on the record again.

The second scenario is while calling it from a different APEX class, typically in a more complex scenario involving multiple REST calls.

We'll look at an example for both situations.

Triggering the call from Process Builder

With the generic class in place we can now trigger the code from Process Builder in the right scenarios.



This shows an example of the code being triggered in a flow based on the Contact record. When calling the Apex you can pass in a single variable from the Process Builder into the Apex method. So, we use a function to combine the values that we want to pass into the Apex from the Process Builder:

Call Apex ?

Action Name * ⓘ

Pega Blacklist Call

Apex Class * ⓘ

Perform REST Call

Set Apex Variables

Field *	Type *	Value *
InputData	Formula ▼	[Contact].Id + '.Blacklist'

Figure 3 Proces Builder Interface

This shows how we pass the Id of the Contact and the string 'Blacklist' (for the method that we want to call, as defined in the Integration Definition admin screen) into the Apex class.

This way for any new call to a REST service we only need to make sure that the relevant information is entered in the administration screen and then we can do the actual call from a process.

Using the RESTCallOut class from Apex

In this scenario we are orchestrating multiple calls, using either the RESTCallOut class or other classes and finally are writing results back to the Salesforce database.

The following is an example where we use the RESTCallOut class in another Apex class. In this example we perform 3 REST Calls:

1. Create a case in an external system, using the RESTCallOut class
2. Add a list of products to that case in the external system, using a custom method
3. Submit the case in the external system, using the RESTCallOut class

The calls in this example are obviously very much related. The first call returns an Id for the case that we need in the next two calls. Also, the external system has a build in security measure that requires you to pass a security code with each update to an case and then returns you a security code for the next call. So, the second call returns us a security code that we need in the third call again.

The complete code for this looks like:

```
@future (callout=true)
public static void SendCaseAndProducts(String RecordId)
{
    List<String> Methods = new List<String> {'CreateCase', 'PutProduct', 'SubmitCase'};
    List<String> IDs = new List<String> {RecordId, '', RecordId};

    Map<String,Object> IntegrationDefs =
    RESTCallOut_V2.LoadIntegrationStructure(Methods);
    System.debug(JSON.serialize(IntegrationDefs));
    RESTCallOut_V2.SetIds(IntegrationDefs, Methods, IDs);
    RESTCallOut_V2.FinalizeObjectQueries(IntegrationDefs);
    RESTCallOut_V2.LoadObjectData(IntegrationDefs);
    System.debug(JSON.serialize(IntegrationDefs));
}
```

```

Map<String,Object> IntDef = (Map<String,Object>)IntegrationDefs.get('CreateCase');

List<Object> DataObjects = (List<Object>)IntDef.get(RESTCallOut_V2.STRUCTURE_DATA);

if (DataObjects.size() > 0)
{
    Map<String, Object> CreateCaseRecord = (Map<String,Object>)DataObjects[0];
    if (RESTCallOut_V2.PerformSingleRESTCall(IntDef, CreateCaseRecord))
    {
        Map<String,Object> SendProductData = new Map<String,Object>();
        SendProductData.put('CaseId', RecordId);
        SendProductData.put('Pega_Case_Id__c',
CreateCaseRecord.get('Pega_Case_Id__c'));
        IntDef = (Map<String,Object>)IntegrationDefs.get('PutProduct');
        if (SendProductsV3(IntDef,SendProductData))
        {
            IntDef = (Map<String,Object>)IntegrationDefs.get('SubmitCase');
            List<Object> SubmitObjects =
(List<Object>)IntDef.get(RESTCallOut_V2.STRUCTURE_DATA);
            if (SubmitObjects.size() > 0)
            {
                Map<String,Object> SubmitObject =
(Map<String,Object>)SubmitObjects[0];
                SubmitObject.put('Pega_Etag__c',
SendProductData.get('Pega_Etag__c'));
                SubmitObject.put('Pega_Case_Id__c',
CreateCaseRecord.get('Pega_Case_Id__c'));
                if (RESTCallOut_V2.PerformSingleRESTCall(IntDef, SubmitObject))
                {
                    IntDef = (Map<String,Object>)IntegrationDefs.get('CreateCase');
                    RESTCallOut_V2.SaveObjectData(IntDef);
                    IntDef = (Map<String,Object>)IntegrationDefs.get('SubmitCase');
                    RESTCallOut_V2.SaveObjectData(IntDef);

                }
            }
        }
    }
}
}
}
}

```

Alternative – Platform Events

The alternative solution would be to use Platform Events, currently in public beta on Salesforce Winter '17 release.

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.

With this Salesforce could simply trigger events in the relevant situations where another system needs to be notified. External systems can listen to the events and retrieve the relevant data for the event from Salesforce.

For a detailed description of the platform events see the beta documentation:

https://resources.docs.salesforce.com/204/latest/en-us/sfdc/pdf/platform_events_beta.pdf

Improvement – Platform Cache

Platform Cache is a memory layer that stores Salesforce session and org data for later access. When you use Platform Cache, your applications can run faster because they store

reusable data in memory. Applications can quickly access this data; they don't need to duplicate calculations and requests to the database on subsequent transactions. In short, think of Platform Cache as RAM for your cloud application.

There are two types of Platform Cache: org cache and session cache. Org cache is a shared cache for all users, session cache is a specific cache for a single user session.

The current solution needs to query Salesforce for each integration call to determine the input and output parameters. But this integration administration is also data that will typically not change very often and is thereby a perfect candidate to place in the platform cache. This is something that we would place in the org cache making it available to all users.

You might be wondering how much performance your app gains by using Platform Cache. Retrieving data from the cache is much faster than through an API call. When comparing SOQL to cache retrieval times, the cache is also much faster than SOQL queries.

The following chart shows the retrieval times, in milliseconds, of data through an API call and the cache. It is easy to notice the huge performance gain when fetching data locally through the cache, especially when retrieving data in multiple transactions. In the sample used for the graph, the cache is hundreds of times faster than API calls. Keep in mind that this chart is a sample test and actual numbers might vary for other apps.

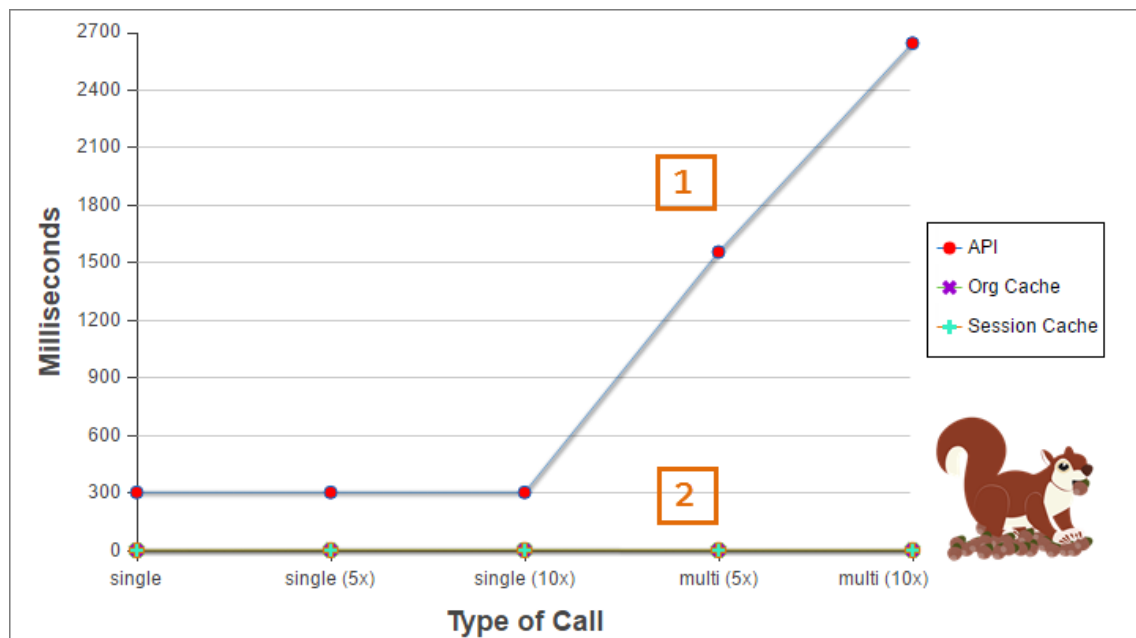


Figure 4 - Making API Calls to External Services Is Slower (1) Than Getting Data from the Cache (2).

This next graph compares SOQL with org and session cache retrieval times. As you can see, SOQL is slower than the cache. In this example, the cache is two or more times faster than SOQL for data retrieval in a single transaction. When performing retrievals in multiple transactions, the difference is even larger. (Note that this graph is a sample and actual numbers might vary for other apps.)

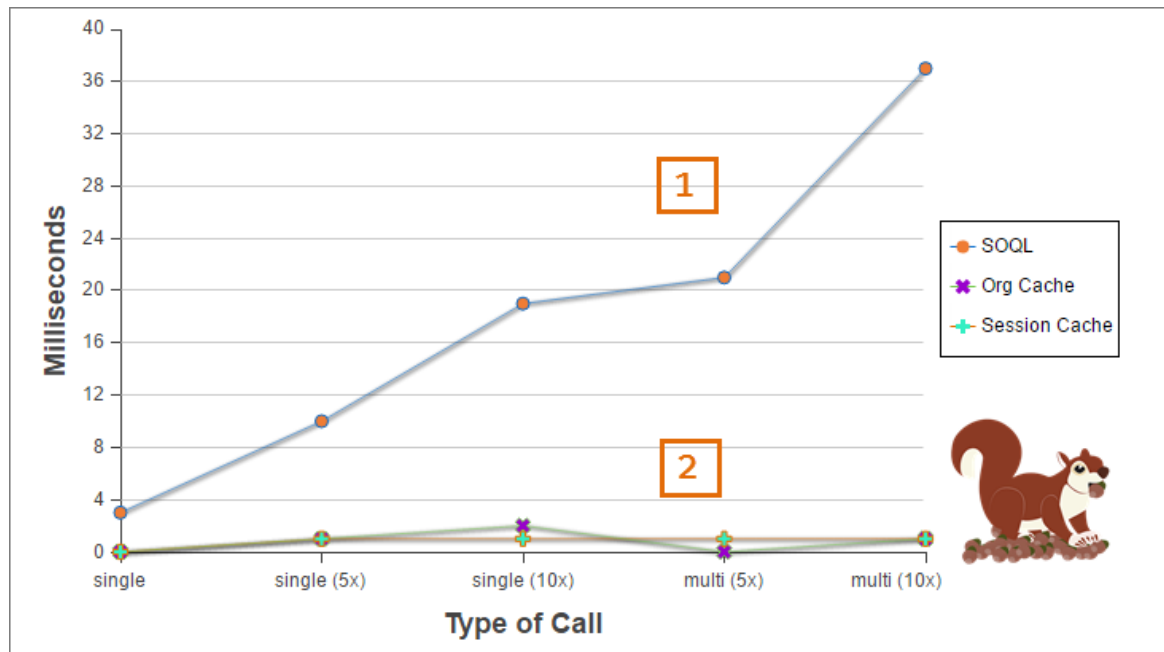


Figure 5 - Getting Data Through SOQL Queries (1) Is Slower Than Getting Data from the Org and Session Cache (2).