TEST DATA CREATION FRAMEWORK

TABLE OF CONTENTS

Overview 3
Classes.....5
Class Structure 6
Script used to create new Json Strings 28
Examples using the Framework 29

OVERVIEW

Some development teams copy and paste code from 1 unit test to another to create test data for their testmethod. Then when they need to add a new field because it is a new required field that is failing tests they realise they've got to copy to all classes.

Some development teams improve this by having a central class but the functions they make have passed in arguments for each field, so for each field that needs to be passed in requires a change to this class.

Some improve this by passing in a list of a key value pair classes.

Some may improve even further by having a for loop to create as many records as you like.

This Test Data Creation Framework has all the above benefits plus uses json to serialise and de-serialise to / from generic sobjects. Uses OOP to provide a logical framework separating functionality that provide only returning Casted Sobjects, inserts single objects, inserts multiple objects for bulk testing, updates objects, inserts multiple objects of different types and creates relationships, and inserts multiple sets of complex linked data structures.

Building into the framework a highly flexible capability so whatever type of test data is required the framework can easily create any data structures.

The framework brings the following benefits:

1. A centralized system making it easy to react to changes in objects.

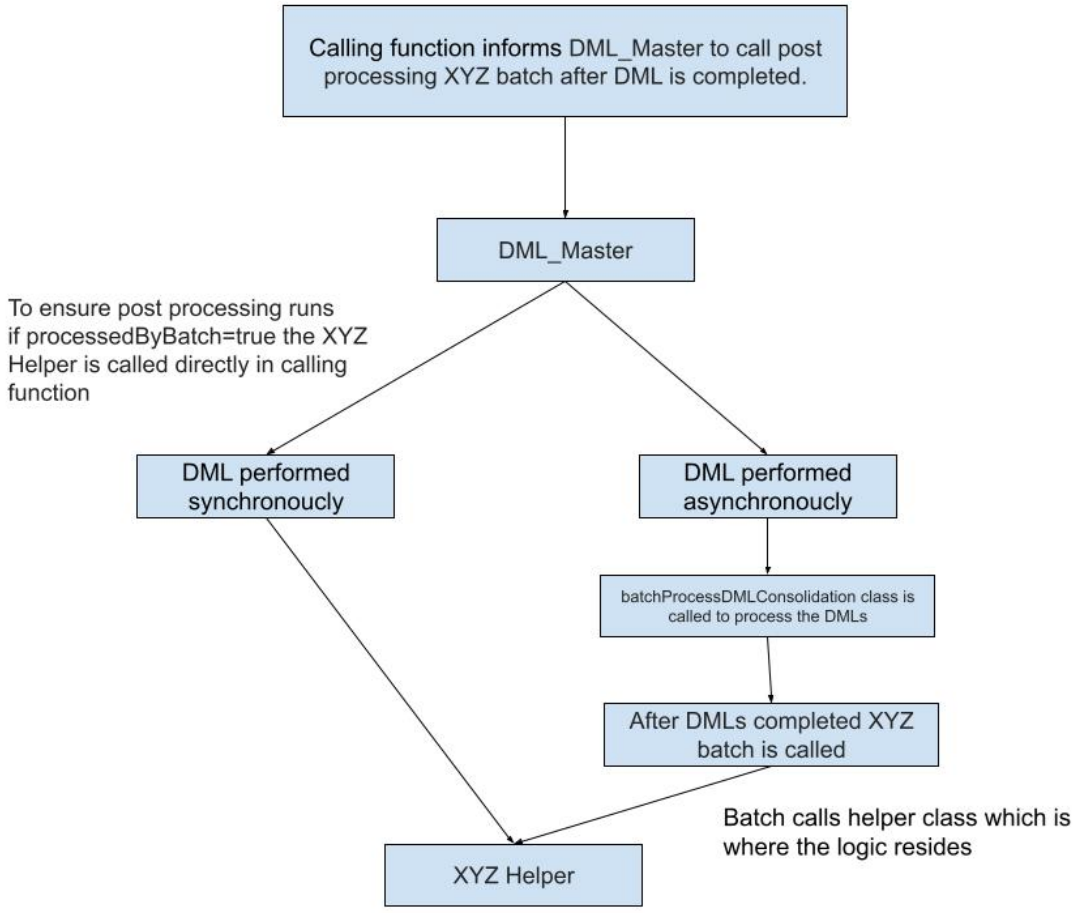
2. The framework is highly flexible, so whatever type of test data is required the framework can easily create.

3. Provides an option for rapid transactional processing by bypassing triggers when creating unit test data to speed up unit tests and deployments.

4. Easy to understand.

5. Ability to automatically perform post processing functionality

   The framework can be used beyond creating data for unit tests; to create data for the running code. By using the DML_Master class all the dml checking that is performed in this class to ensure safe creation of data can be used as an utility class for running code. Optionally, you can command the DML_Master class to automatically perform post processing functionality, such as to command the system to run a bespoke function to reconcile data sent to 3rd party systems to ensure this was successful. The DML_Master class will autonomously decide if the post processing is performed synchronously or asynchronously. The following diagram depicts how this process occurs.



CLASSES

So, let's get started.

First create several classes

ITestData
: This is the interface class

TestDataJsonLibrary
: Provides the json strings that will be serialised into sobjects

TestDataReturn
: Serialises json strings from TestDataJsonLibrary or custom json strings

TestDataCoreInsert
: Main insert functions

TestDataInsertObjects
: DML transactions of the serialised json strings occurs here, but only inserts 1 record

TestDataUpdateObjects
: Inserts and then updates a record

TestDataRelatedObjects
: Inserts multiple records of different Sobjects and relate the records together

TestDataBulkObjects
: Inserts multiple records

DML_Master
: A utility class used to assign values to any fields of any Sobject and to commit DMLs to the database

UtilityObjectData
: Used to override any field values

BatchDMLPostProcessorInterface
: Interface class to process batchProcessDMLConsolidation batch

batchProcessDMLConsolidation
: If the dml cannot be processed in real time due to governor limits this class creates the records asynchronously to avoid breaking governor limits

GenericDMLResult

This class holds details of any DML records such as failures etc

Optional:

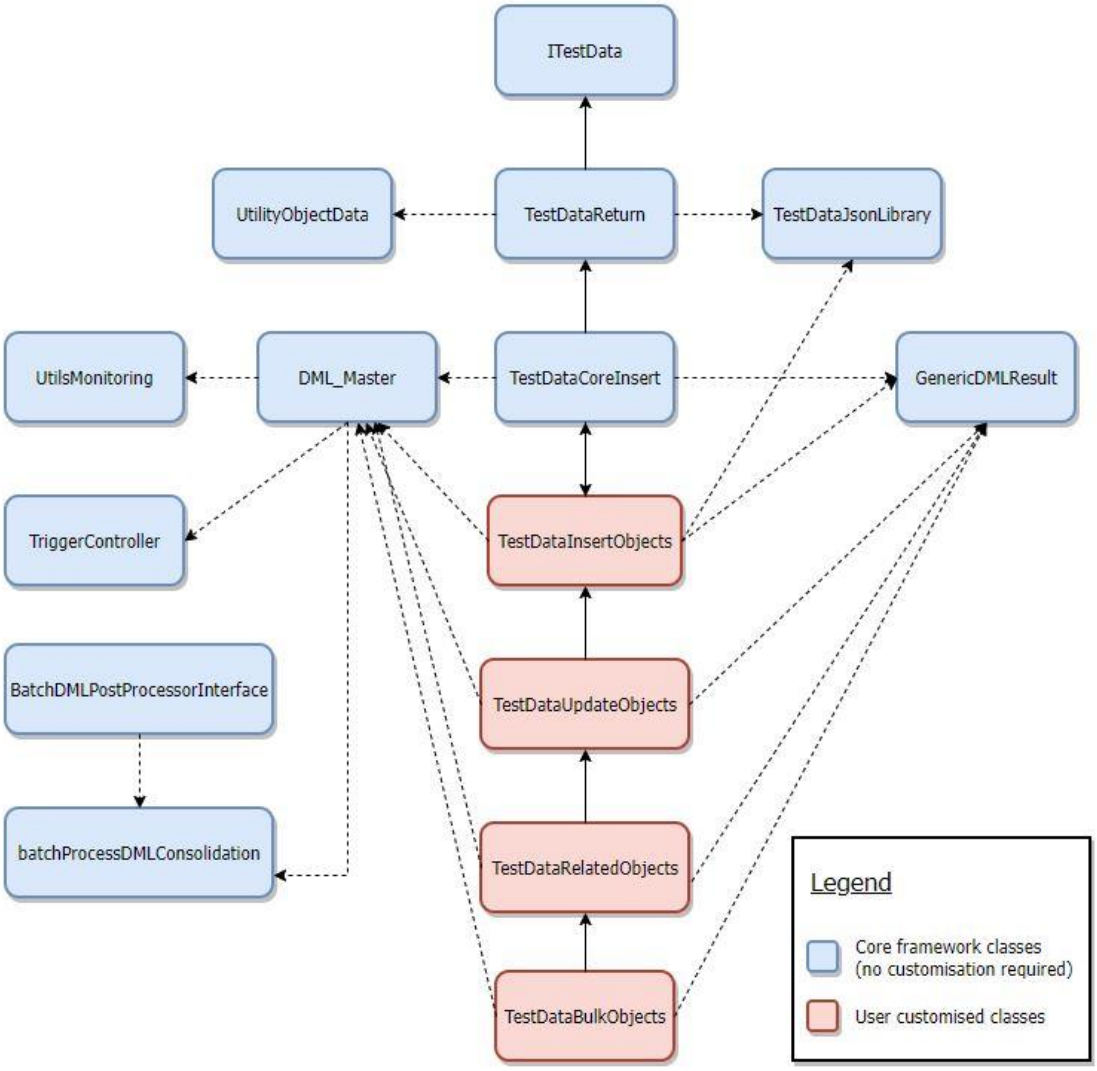
UtilsMonitoring

This class processes any error messages and records them in an object

TriggerController

Used to bypass triggers. Note, this class is not included in this framework because it involves implementing a different framework first that controls bypassing triggers and because a different trigger framework has been adopted this isn't included. Therefore, the DML_Master class will need to be altered since this class is referenced several times.

Class Structure



The following code may not appear indented correctly but do not be concerned once you copy the code into your apex classes the code will be properly indented.

In the ITestData class add

```
public interface ITestData {

List<sObject> returnAnyObject(String jsonStr, Map<String, Object> fldVals);

}
```

Next, in the TestDataJsonLibrary class add

```
public class TestDataJsonLibrary {

public class Standard{
public Map<String, String> stdJsonMap = new Map<String, String>();

for (Test_Data__mdt m : [Select Label,Json__c From Test_Data__mdt where Category__c='Standard']){
            stdJsonMap.put(m.Label, m.ses5__Json__c);
        }
}
}
```

Next, in the TestDataReturn class add

```
public abstract class TestDataReturn implements ITestData{
public List<sObject> returnAnyObject(String jsonStr, Map<String, Object> fldVals) {
    //json made correctly
    if (!jsonStr.startsWith('[')) {
        jsonStr = '[' + jsonStr + ']';
    }

    //deserialize JSON
    List<SObject> sObj = (List<SObject>) System.JSON.deserialize(jsonStr, List<SObject>.class);

    //set other field values
    if (fldVals != null) {
        for (SObject so : sObj) {
            so = UtilityObjectData.setSObjectFieldValues(so, fldVals);
        }
    }
```

```
        return sObj;
    }
}
```

Next, in the TestDataCoreInsert class add

```
public abstract with sharing class TestDataCoreInsert extends TestDataReturn{
    public Map<System.Type, String> useThisJson = new Map<System.Type, String>();
    public Boolean bulkInsert = false;
    public Boolean saveErrors = false;
    public String debugMessageType = 'Test Data Framework';
    public Boolean allowBatch = false;
    public Integer batchSize = 200;
    public String deactivateTrigger = TriggerController.TRIGGER_ALL;
    public String postProcessorClass;

    public sObject insertAnyObject(String thisjson, Map<String, Object> fldVals, GenericDMLResult[]
dbErrors, System.Type thisObjType){
        // is a different json been set
        if(this.useThisJson != null && this.useThisJson.containsKey(thisObjType))
            thisjson = this.useThisJson.get(thisObjType);

        //pass json to object creator -  no DML
        sObject sobj = super.returnAnyObject(thisjson, fldVals)[0];

        // if bulkInsert is on then just return this 1 object which will be passed to Bulk Insert class to insert
a number of objects
        //otherwise if False do insert object
        if(!bulkInsert)
            DML_Master.insertObjects(
                new SObject[]{ sobj },
                this.allowBatch,        // allow batch
                this.batchSize,         // batch size
                dbErrors,               // list to save errors into
                this.saveErrors,        // save errors
                this.debugMessageType,  // message type
                this.deactivateTrigger,
                this.postProcessorClass // batch post processor
            );

        return sobj;
    }

}
```

Next, in the TestDataInsertObjects class add

```

```

```
public virtual with sharing class TestDataInsertObjects extends TestDataCoreInsert {

public xxx__c insertXXX(Map<String, Object> fldVals, GenericDMLResult[] errors){
return (xxx__c) insertAnyObject(new TestDataJsonLibrary.Standard().stdJsonMap.get('xxx__c'), fldVals,
errors, xxx__c.class);
}

public Account[] insertBulkAccounts(Map<String, Object> fldVals, GenericDMLResult[] errors){
return (Account[]) insertAllObjects(new TestDataJsonLibrary.Standard().stdJsonMap.get('Account'), fldVals,
errors, Account.class);
}

}
```

Next, in the TestDataUpdateObjects class add

```
public virtual with sharing class TestDataUpdateObjects extends TestDataRelatedObjects{

public xxx__c updateXXX(Map<String, Object> fldIns, Map<String, Object> fldUpd, GenericDMLResult[]
errors){
xxx__c xxx = super.insertSpoke(fldIns, errors);

      //set any additional fields
      if(fldUpd != null && !fldUpd.isempty())
         xxx = (xxx__c)(UtilityObjectData.setSObjectFieldValues((Sobject)xxx, fldUpd));

          DML_Master.updateObjects(
               new SObject[]{ xxx },
               super.allowBatch,        // allow batch
               super.batchSize,         // batch size
               errors,              // list to save errors into
               super.saveErrors,        // save errors
               super.debugMessageType,  // message type
               super.deactivateTrigger,
               super.postProcessorClass // batch post processor
);
      return xxx;
}
}
```

Next, in the TestDataRelatedObjects class add

```
public virtual with sharing class TestDataRelatedObjects extends TestDataInsertObjects{

public Lead xxx;

public Account insertAccountAndLead(Map<System.Type, Map<String, Object>> kMaps,
GenericDMLResult[] dbErrors){
    if(kMaps == null){
        kMaps = new Map<System.Type, Map<String, Object>>();
    }

    if (!kMaps.containskey(Lead.class))
            kMaps.put(Lead.class, new Map<String, Object>());

    if (!kMaps.containskey(Account.class))
kMaps.put(Account.class, new Map<String, Object>());

xxx = super.insertLead(kMaps.get(Lead.class), dbErrors);

if (!kMaps.containskey(Account.class)) kMaps.put(Account.class, new Map<String, Object>());
kMaps.get(Account.class).put('Lead',xxx.id);

return super.insertAccount(kMaps.get(Account.class), dbErrors);
}

}
```

Next, in the TestDataBulkObjects class add

```
public virtual with sharing class TestDataBulk extends TestDataRelatedObjects{

    public SObject[] insertBulkObjects(Integer recordCount, String jsonStr, Map<String, Object> fieldValues,
                                       GenericDMLResult[] errors, System.Type objType) {
        SObject[] sObj = new SObject[]{};
        //  prevents the records from being inserted individually
        bulkInsert = true;

        // create the desired number of records
        for (Integer i = 1; i <= recordCount; i++) {
            // adds a new record to the list of objects
            sObj.add(super.insertAnyObject(jsonStr, fieldValues, errors, objType));
        }

        // perform standard insert on the list of objects
        DML_Master.insertObjects(
                sObj,
                super.allowBatch,        // allow batch
                super.batchSize,         // batch size
                errors,              // list to save errors into
                super.saveErrors,        // save errors
                super.debugMessageType,  // message type
                super.deactivateTrigger,
                super.postProcessorClass // batch post processor
        );

        bulkInsert = false;

        return sObj;
    }

    public SObject[] insertBulkObjects(Integer recordCount, String jsonStr, Map<integer,Map<String,
Object>> fieldValues,
                                       GenericDMLResult[] errors, System.Type objType) {
        SObject[] sObj = new SObject[]{};

        //  prevents the records from being inserted individually
        bulkInsert = true;

        // create the desired number of records
        for (Integer i = 1; i <= recordCount; i++) {
            // adds a new record to the list of objects
            if (fieldValues.containskey(i)){
                sObj.add(super.insertAnyObject(jsonStr, fieldValues.get(i), errors, objType));
            }
        }
```

```
        // perform standard insert on the list of objects
        DML_Master.insertObjects(
                sObj,
                super.allowBatch,        // allow batch
                super.batchSize,         // batch size
                errors,                  // list to save errors into
                super.saveErrors,        // save errors
                super.debugMessageType,  // message type
                super.deactivateTrigger,
                super.postProcessorClass // batch post processor
        );

        bulkInsert = false;

        return sObj;
    }

    public Account[] insertAccounts(Integer recordCount, Map<String, Object> fieldValues,
GenericDMLResult[] errors) {
        return (Account[]) insertBulkObjects(
                recordCount,
                new TestDataJsonLibrary.Standard().stdJsonMap.get(Constant.CONST_Account),
                fieldValues, errors, Account.class
        );
    }

    public Account[] insertAccounts(Integer recordCount, Map<integer,Map<String, Object>> fieldValues,
GenericDMLResult[] errors) {
        return (ses__Object_ID__c[]) insertBulkObjects(
                recordCount,
                new TestDataJsonLibrary.Standard().stdJsonMap.get(Constant.CONST_Account),
                fieldValues, errors, Account.class
        );
    }

public Lead insertLead(Integer recordCount, Map<integer,Map<String, Object>> fieldValues,
GenericDMLResult[] errors) {
        return (Lead) insertBulkObjects(
                recordCount,
                new TestDataJsonLibrary.Standard().stdJsonMap.get('Lead'),
                fieldValues, errors, Lead.class
        );
    }

}
```

Next, in the DML_Master class add

```
public abstract class DML_Master {

public enum dmlType
{INSERT_OBJECT,UPDATE_OBJECT,UPSERT_OBJECT,DELETE_OBJECT,UNDELETE_OBJECT}

public class DMLMasterException extends Exception { }

public static integer defaultBatchQuantity = 200;

public static Boolean processedByBatch = false;

public static final String TRIGGER_NONE = 'None';
public static final String TRIGGER_ALL = 'ALL';
public static final String TRIGGER_INSERT = 'INSERT';
public static final String TRIGGER_UPDATE = 'UPDATE';
public static final String TRIGGER_DELETE = 'DELETE';
public static final String TRIGGER_UNDELETE = 'UNDELETE';

public  static final String NO_CREATE_PERMISSION        = 'Has No Create Permission';
public  static final String NO_UPDATE_PERMISSION        = 'Has No Update Permission';
public  static final String NO_DELETE_PERMISSION        = 'Has No Delete Permission';
public  static final String NO_UNDELETE_PERMISSION      = 'Has No UnDelete Permission';


public static GenericDMLResult[] masterErrorObject;

private static String defaultMsgCategory = 'Test Data';

public  static boolean insertObjects(sObject[] sObj){
//this is the default insert function which allows the user to pass fewer arguments
masterErrorObject = new GenericDMLResult[]{};
return genericDML(sObj, true, 200, masterErrorObject, true, dmlType.INSERT_OBJECT, 'Insert Objects',
TRIGGER_NONE, null);
}

public  static boolean insertObjects(sObject[] sObj, Boolean processByBatch, Integer batchQuantity,
GenericDMLResult[] errors, Boolean saveErrors, String msgCategory, String trigOff, String
postProcessingClass){
return genericDML(sObj, processByBatch, batchQuantity, errors, saveErrors, dmlType.INSERT_OBJECT,
msgCategory, trigOff, postProcessingClass);
}

public  static boolean insertBatchObjects(sObject[] sObj, Boolean processByBatch, Integer batchQuantity,
GenericDMLResult[] errors, Boolean saveErrors, String msgCategory, String trigOff, String
postProcessingClass){
return processRecordsByBatch(sObj, processByBatch, batchQuantity, errors, saveErrors,
dmlType.INSERT_OBJECT, msgCategory, trigOff, postProcessingClass);
```

```
}

public  static boolean updateObjects(sObject[] sObj){
masterErrorObject = new GenericDMLResult[]{};
return genericDML(sObj, true, 200, masterErrorObject, true, dmlType.UPDATE_OBJECT,
defaultMsgCategory, TRIGGER_NONE, null);
}

public  static boolean updateObjects(sObject[] sObj, Boolean processByBatch, Integer batchQuantity,
GenericDMLResult[] errors, Boolean saveErrors, String msgCategory, String trigOff, String
postProcessingClass){
return genericDML(sObj, processByBatch, batchQuantity, errors, saveErrors, dmlType.UPDATE_OBJECT,
msgCategory, trigOff, postProcessingClass);
}

public  static boolean updateBatchObjects(sObject[] sObj, Boolean processByBatch, Integer batchQuantity,
GenericDMLResult[] errors, Boolean saveErrors, String msgCategory, String trigOff, String
postProcessingClass){
return processRecordsByBatch(sObj, processByBatch, batchQuantity, errors, saveErrors,
dmlType.UPDATE_OBJECT, msgCategory, trigOff, postProcessingClass);
}

public  static boolean deleteObjects(sObject[] sObj){
masterErrorObject = new GenericDMLResult[]{};
return genericDML(sObj, true, 200, masterErrorObject, true, dmlType.DELETE_OBJECT,
defaultMsgCategory, TRIGGER_NONE, null);
}

public  static boolean deleteObjects(sObject[] sObj, Boolean processByBatch, Integer batchQuantity,
GenericDMLResult[] errors, Boolean saveErrors, String msgCategory, String trigOff, String
postProcessingClass){
return genericDML(sObj, processByBatch, batchQuantity, errors, saveErrors, dmlType.DELETE_OBJECT,
msgCategory, trigOff, postProcessingClass);
}

public  static boolean deleteBatchObjects(sObject[] sObj, Boolean processByBatch, Integer batchQuantity,
GenericDMLResult[] errors, Boolean saveErrors, String msgCategory, String trigOff, String
postProcessingClass){
return processRecordsByBatch(sObj, processByBatch, batchQuantity, errors, saveErrors,
dmlType.DELETE_OBJECT, msgCategory, trigOff, postProcessingClass);
}

public  static boolean undeleteObjects(sObject[] sObj){
masterErrorObject = new GenericDMLResult[]{};
return genericDML(sObj, true, 200, masterErrorObject, true, dmlType.UNDELETE_OBJECT,
defaultMsgCategory, TRIGGER_NONE, null);
}

public  static boolean undeleteObjects(sObject[] sObj, Boolean processByBatch, Integer batchQuantity,
GenericDMLResult[] errors, Boolean saveErrors, String msgCategory, String trigOff, String
postProcessingClass){
```

```
return genericDML(sObj, processByBatch, batchQuantity, errors, saveErrors,
dmlType.UNDELETE_OBJECT, msgCategory, trigOff, postProcessingClass);
}

public  static boolean genericDML(sObject[] sObj, Boolean processByBatch, Integer batchQuantity,
GenericDMLResult[] errors, Boolean saveErrors, dmlType thisDMLType, String setErrorMsg, String trigOff,
String postProcessingClass) {
Boolean errorsOccurred = false;
processedByBatch = false;
String errorMessage;
try{
        if (thisDMLType == null || sObj == null) {
            throw new DMLMasterException('genericDML: Neither DmlType nor sObject[] can be NULL');
        }

        if (sObj.size() == 1 && sObj[0] == null) {
            throw new DMLMasterException('genericDML: sObject[] cannot contain NULL');
        }

        if (sObj.isEmpty()) {
            return true;
        }

Schema.DescribeSObjectResult objDescribe = sObj[0].getSObjectType().getDescribe();
UtilsMonitoring.setupMonitoring();
        // check we have permission
        if (!checkPermissionForDML(thisDMLType, objDescribe)) {
             String msgPermis;
             if (thisDMLType == dmlType.INSERT_OBJECT) {
                        msgPermis = NO_CREATE_PERMISSION;
             }
             else if (thisDMLType == DmlType.UPDATE_OBJECT){
                        msgPermis = NO_UPDATE_PERMISSION;
             }
             else if (thisDMLType == DmlType.DELETE_OBJECT){
                        msgPermis = NO_DELETE_PERMISSION;
             }
             else if (thisDMLType == DmlType.UNDELETE_OBJECT){
                        msgPermis = NO_UNDELETE_PERMISSION;
             }
            errorMessage = msgPermis + thisDMLType + ' ' + objDescribe.getName();
            handleError(errorMessage, errors, saveErrors, setErrorMsg);
            errorsOccurred = true;
        }
        else {
             //permission allowed for dml
        Integer dmlCount = 1;
        Integer recordCount = sObj.size();

            // to support UPSERT, create separate lists for INSERT and UPDATE
            SObject[] sObjInsert;
            SObject[] sObjUpdate;
```

```
            if (thisDMLType == DmlType.INSERT_OBJECT) {
                sObjInsert = sObj;
            }
            if (thisDMLType == DmlType.UPDATE_OBJECT) {
                sObjUpdate = sObj;
            }
            if (thisDMLType == DmlType.UPSERT_OBJECT) {
                sObjUpdate = new SObject[]{};
                sObjUpdate.addAll(sObj);
                sObjInsert = UtilityObjectData.extractNewRecords(sObjUpdate);
                if (!sObjInsert.isEmpty() && !sObjUpdate.isEmpty()) {
                    dmlCount = 2;
                }
            }

Type convertedType = Type.forname(String.valueOf(sObj[0].getsObjectType()));
if (trigOff != TRIGGER_NONE){
//this allows you to bypass triggers to improve efficiency of processing
if (trigOff == TRIGGER_ALL)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_NONE, true);
else if (trigOff == TRIGGER_INSERT)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_INSERT, true);
else if (trigOff == TRIGGER_UPDATE)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_UPDATE, true);
else if (trigOff == TRIGGER_DELETE)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_DELETE, true);
else if (trigOff == TRIGGER_UNDELETE)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_UNDELETE, true);
}

//check number of total dml statements and number of records in dml
if ((Limits.getDmlStatements() <= (Limits.getLimitDmlStatements() - dmlCount)) && (Limits.getDmlRows() +
recordCount <= Limits.getLimitDmlRows())){

if (errors != null){
Database.saveresult[] res;
if (thisDMLType == dmlType.INSERT_OBJECT) {
res = Database.insert(sObjInsert,false);
                GenericDMLResult[] genericResults = GenericDMLResult.makeGeneric(res, sObjInsert);
                errors.addAll(getDMLErrors(DML_Master.class,genericResults, saveErrors, setErrorMsg));
}
else if (thisDMLType == dmlType.UPDATE_OBJECT){
res = Database.update(sObjUpdate,false);
                GenericDMLResult[] genericResults = GenericDMLResult.makeGeneric(res, sObjUpdate);
                errors.addAll(getDMLErrors(DML_Master.class,genericResults, saveErrors, setErrorMsg));
}
else if (thisDMLType == dmlType.DELETE_OBJECT){
Database.Deleteresult[] desRes = Database.delete(sObj,false);
                GenericDMLResult[] genericResults = GenericDMLResult.makeGeneric(desRes, sObj);
                errors.addAll(getDMLErrors(DML_Master.class, genericResults, saveErrors, setErrorMsg));
}
else if (thisDMLType == dmlType.UNDELETE_OBJECT){
```

```
Database.Undeleteresult[] undesRes = Database.undelete(sObj,false);
                GenericDMLResult[] genericResults = GenericDMLResult.makeGeneric(undesRes, sObj);
                errors.addAll(getDMLErrors(DML_Master.class, genericResults, saveErrors, setErrorMsg));
}
}
else{
            if ((thisDMLType == DmlType.INSERT_OBJECT)
                || (thisDMLType == DmlType.UPSERT_OBJECT && !sObjInsert.isEmpty())) {
insert sObjInsert;
}
            if ((thisDMLType == DmlType.UPDATE_OBJECT)
                || (thisDMLType == DmlType.UPSERT_OBJECT && !sObjUpdate.isEmpty())) {
update sObjUpdate;
}else if (thisDMLType == dmlType.DELETE_OBJECT){
delete sObj;
}else if (thisDMLType == dmlType.UNDELETE_OBJECT){
undelete sObj;
}
}

}
else{
//would break governor limits so try by batch
if (processByBatch){
processedByBatch = true;
processRecordsByBatch(sObj, processByBatch, batchQuantity, errors, saveErrors, thisDMLType,
setErrorMsg, trigOff, postProcessingClass);
}
else{
UtilsMonitoring.buildMonitoringMessage(DML_Master.class, setErrorMsg, 'Could not save records.
Processing by batch was not allowed.', null);
}
}

if (trigOff != TRIGGER_NONE){
//this resets trigger control
if (trigOff == TRIGGER_ALL)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_NONE, false);
else if (trigOff == TRIGGER_INSERT)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_INSERT, false);
else if (trigOff == TRIGGER_UPDATE)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_UPDATE, false);
else if (trigOff == TRIGGER_DELETE)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_DELETE, false);
else if (trigOff == TRIGGER_UNDELETE)
TriggerController.setTriggerControlValue(convertedType, TriggerController.TRIGGER_UNDELETE, false);
}
//End permission allowed for dml
        }
}
catch(Exception ex){
        errorMessage = MSG_EXCEPTION + ex.getMessage();
```

```
        handleError(errorMessage, errors, saveErrors, setErrorMsg);
        errorsOccurred = true;
    }
    finally {
        if (errorsOccurred && saveErrors) {
            UtilsMonitoring.saveMonitoringMessages(DML_Master.class);
        }
    }

    return !errorsOccurred;
}

public  static Boolean processRecordsByBatch(sObject[] sObj, Boolean processByBatch, Integer
batchQuantity, GenericDMLResult[] errors, Boolean saveErrors, dmlType thisDMLType, String setErrorMsg,
String trigOff, String postProcessingClass) {
Boolean errorsOccurred = false;
if (!System.isBatch() && !System.isFuture()) {
Database.executebatch(new batchProcessDMLConsolidation(thisDMLType, sObj, (errors == null),
saveErrors, setErrorMsg, trigOff, postProcessingClass), (batchQuantity != null) ? batchQuantity : 200);
}else{
UtilsMonitoring.buildMonitoringMessage(DML_Master.class, setErrorMsg, 'Could not save records.
Processing by batch was not allowed as context is already in batch or a future.', null);
errorsOccurred = true;
}

return !errorsOccurred;
}



    @TestVisible private static void handleError(String errorMessage, GenericDMLResult[] dmlResults,
Boolean saveErrors, String messageType) {
        if (dmlResults != null) {
            dmlResults.add(new GenericDMLResult(false, null, null, errorMessage));
        }
        if (saveErrors) {
            UtilsMonitoring.buildMonitoringMessage(DML_Master.class, messageType, errorMessage, null);
        }
    }

    @TestVisible
    private static Boolean checkPermissionForDML(DmlType thisDMLType, Schema.DescribeSObjectResult
objDescribe) {
        return (
                (thisDMLType == DmlType.INSERT_OBJECT && objDescribe.isCreateable())
                        || (thisDMLType == DmlType.UPDATE_OBJECT && objDescribe.isUpdateable())
                        || (thisDMLType == DmlType.UPSERT_OBJECT && objDescribe.isCreateable() &&
objDescribe.isUpdateable())
                        || (thisDMLType == DmlType.DELETE_OBJECT && objDescribe.isDeletable())
                        || (thisDMLType == DmlType.UNDELETE_OBJECT && objDescribe.isUndeletable())
        );
    }
```

```
    @TestVisible private static GenericDMLResult[] getDMLErrors(System.Type cls, GenericDMLResult[]
dmlResults, Boolean saveErrors, String messageType){
        GenericDMLResult[] dmlErrors = new GenericDMLResult[]{};

        // look through all the results for any failures
        for (GenericDMLResult result : dmlResults) {
            if (!result.success) {
                // operation failed, so keep this result record
                dmlErrors.add(result);
                // report the errors
                if (saveErrors) {
                    for (GenericDMLResult.GenericError err : result.errors) {
                        String errorMsg = err.statusCode + ': ' + err.message;
                        UtilsMonitoring.buildMonitoringMessage(cls, messageType, errorMsg, null);
                    }
                }
            }
        }

        return dmlErrors;
    }
}
```

Next, in the UtilityObjectData class add

```
public with sharing class UtilityObjectData {
private class TooManyBatchApexJobsException extends Exception{}

       public static Sobject setSObjectFieldValues(Sobject aobj, Map<String, Object> fldVals){
        if (aobj != null && fldVals != null) {
            for (String field : fldVals.keySet()) {
                aobj.put(field, fldVals.get(field));
            }
        }
        return aobj;
       }

    public static SObject[] extractNewRecords(SObject[] sourceRecords) {
            //extracts records for insert only
        SObject[] newRecords = new SObject[]{};
        Integer i = 0;
        while (i < sourceRecords.size()) {
            if (sourceRecords[i].Id == null) {
                newRecords.add(sourceRecords.remove(i));
            }
            else {
                i++;
            }
```

```
        }
        return newRecords;
    }
}
```

Next, in the batchProcessDMLConsolidation class add

```
global class batchProcessDMLConsolidation implements Database.Batchable<sObject>, Database.Stateful
{
global Sobject[] allSobj;
global DML_Master.DmlType dml_Type;
    global GenericDMLResult[] dmlErrors;
    global Boolean allOrNone;
    global Boolean saveErrors;
    global String messageType;
    global String trigOff;
    global String postProcessingClass;

global batchProcessDMLConsolidation(DML_Master.DmlType operation, SObject[] records, Boolean
allOrNone, Boolean saveErrors, String messageType, String trigOff, String postProcessingClass) {
        this.allSobj = records;
        this.dml_Type = operation;
        this.allOrNone = allOrNone;
        this.saveErrors = saveErrors;
        this.dmlErrors = new GenericDMLResult[]{};
        this.messageType = messageType;
        this.trigOff = trigOff;
        this.postProcessingClass = postProcessingClass;
    }

global list<Sobject> start(Database.BatchableContext BC) {
return allSobj;
}

global void execute(Database.BatchableContext BC, list<Sobject> sobj) {
        try {
            GenericDMLResult[] batchResults;
            if (!this.allOrNone) {
                batchResults = new GenericDMLResult[]{};
            }

            //call genericDML to do the dml because it could be another function call this batch not necessarily
DML_Master so permissions etc may not have been checked
            Boolean success = DML_Master.genericDML(sobj, false, 0, batchResults, this.saveErrors,
this.dml_Type, this.messageType, trigOff, null);
            if (success){
            if (batchResults != null) {
                this.dmlErrors.addAll(batchResults);
            }
```

```
        }
        catch (Exception ex) {
            UtilsMonitoring.buildMonitoringMessage(batchProcessDMLConsolidation.class, errorCatergory,
'This batch failed.', null);
        }

}

global void finish(Database.BatchableContext BC) {
        if (postProcessingClass != null && String.isNotBlank(postProcessingClass)) {
            BatchDMLPostProcessorInterface postProcessor =
(BatchDMLPostProcessorInterface)Type.forName(postProcessingClass).newInstance();

            postProcessor.performPostProcessing(this);
        }
        UtilsMonitoring.saveMonitoringMessages(batchProcessDMLConsolidation.class);
}
}
```

Next, in the BatchDMLPostProcessorInterface class add

```
public interface BatchDMLPostProcessorInterface {

    void performPostProcessing( batchProcessDMLConsolidation batchDML );

}
```

Next, in the GenericDMLResult class add

```
global class GenericDMLResult {
```

```
    global Id recordId;
    global SObject sObj;
    global Boolean success;//if the DML operation on this record was successful
    global GenericError[] errors;//list of errors for the DML operation

    global GenericDMLResult(Boolean isSuccess, Id recId, SObject recObject) {
        this.success = isSuccess;
        this.recordId = recId;
        this.sObj = recObject;
        this.errors = new GenericError[]{};
    }

    global GenericDMLResult(Boolean isSuccess, Id recId, SObject recObject, String errorMessage) {
        this(isSuccess, recId, recObject);
        if (errorMessage != null) {
            this.errors.add(new GenericError(errorMessage, null, null));
        }
    }

    global GenericDMLResult(Boolean dmlSuccess, Id dmlRecordId, SObject dmlObject, Database.Error[]
dmlErrors) {
        this(dmlSuccess, dmlRecordId, dmlObject);
        if (dmlErrors != null) {
            for (Database.Error dmlError : dmlErrors) {
                this.errors.add(new GenericError(dmlError));
            }
        }
    }

    global class GenericError {
        global String message;
        global String fields;
        global StatusCode statusCode;

        global GenericError(String errorMessage, String errorFields, StatusCode errorCode) {
            this.message = errorMessage;
            this.fields = errorFields;
            this.statusCode = errorCode;
        }

        global GenericError(Database.Error dmlError) {
            this(dmlError.getMessage(), String.join(dmlError.getFields(),','), dmlError.getStatusCode());
        }
    }

    global static GenericDMLResult[] makeGeneric(Database.SaveResult[] results, SObject[] records) {
            //Converts SaveResult[] to GenericDmlResult[]
            //INSERT / UPDATE results as a SaveResult[]
        GenericDMLResult[] genericResults = new GenericDMLResult[]{};
        for (Integer i=0; i<results.size(); i++) {
```

```
            genericResults.add(new GenericDMLResult(results[i].isSuccess(), results[i].getId(), records[i],
results[i].getErrors()));
        }
        return genericResults;
    }

    global static GenericDMLResult[] makeGeneric(Database.UpsertResult[] results, SObject[] records) {
            //Converts UpsertResult[] to GenericDmlResult[]
            //UPSERT results as a UpsertResult[]
        GenericDMLResult[] genericResults = new GenericDMLResult[]{};
        for (Integer i=0; i<results.size(); i++) {
            genericResults.add(new GenericDMLResult(results[i].isSuccess(), results[i].getId(), records[i],
results[i].getErrors()));
        }
        return genericResults;
    }

    global static GenericDMLResult[] makeGeneric(Database.DeleteResult[] results, SObject[] records) {
            //Converts DeleteResult[] to GenericDmlResult[]
            //DELETE results as a DeleteResult[]
        GenericDMLResult[] genericResults = new GenericDMLResult[]{};
        for (Integer i=0; i<results.size(); i++) {
            genericResults.add(new GenericDMLResult(results[i].isSuccess(), results[i].getId(), records[i],
results[i].getErrors()));
        }
        return genericResults;
    }

    global static GenericDMLResult[] makeGeneric(Database.UndeleteResult[] results, SObject[] records) {
            //Converts UndeleteResult[] to GenericDmlResult[]
            //UNDELETE results as a UndeleteResult[]
        GenericDMLResult[] genericResults = new GenericDMLResult[]{};
        for (Integer i=0; i<results.size(); i++) {
            genericResults.add(new GenericDMLResult(results[i].isSuccess(), results[i].getId(), records[i],
results[i].getErrors()));
        }
        return genericResults;
    }
}
```

Next, in the UtilsMonitoring class add

```
public with sharing class UtilsMonitoring {
```

```
public static Set< String > monitorSet;
public static Map<System.Type, Map<String, list<ErrorMessage>>> saveMonitoringMessagesMap;

public class ErrorMessage{
public String msg;
public Map<String, Object> additionalFields;

public ErrorMessage(String aMsg, Map<String, Object> aKVals){
          if (aMsg != null) {
             this.msg = aMsg.abbreviate(255);
          }
this.additionalFields = aKVals;
}
}

public static Boolean getMonitoringCoverage(){
return (((MonitoringCoverage__c.getOrgDefaults() != null) ?
MonitoringCoverage__c.getOrgDefaults().value__c : false) ||
MonitoringCoverage__c.getInstance(UserInfo.getUserId()).value__c  ||
MonitoringCoverage__c.getInstance(UserInfo.getProfileId()).value__c) ;
}

public static void setupMonitoring(){
if (monitorSet == null || monitorSet.isempty()){
Map<String, Monitoring__c> mon = Monitoring__c.getAll();
if (!mon.isempty()){
monitorSet = new Set< String >();
for (Monitoring__c thisMon: mon.Values()){
if ((thisMon.Active__c && thisMon.Monitor_Datetime_From__c == null && thisMon.Monitor_Datetime_To__c
== null) || (thisMon.Active__c && thisMon.Monitor_Datetime_From__c >= Datetime.Now() &&
thisMon.Monitor_Datetime_To__c <= Datetime.Now())){
if (thisMon.Name != null) monitorSet.add(thisMon.Name);
}
}
if (!monitorSet.isempty() && saveMonitoringMessagesMap == null){
saveMonitoringMessagesMap = new Map<System.Type, Map<String, list<ErrorMessage>>>();
}
}
}
}

public static void buildMonitoringMessage(System.Type objMonitor, String ref, String msg, Map<String,
Object> otherFields) {

if (getMonitoringCoverage() || (monitorSet != null && !monitorSet.isempty() && monitorSet.contains(ref))){
if (saveMonitoringMessagesMap != null && saveMonitoringMessagesMap.containskey(objMonitor)){
if (saveMonitoringMessagesMap.get(objMonitor).containskey(ref))
(saveMonitoringMessagesMap.get(objMonitor).get(ref)).add(new ErrorMessage(msg, otherFields));
else{
saveMonitoringMessagesMap.get(objMonitor).put(ref, new list<ErrorMessage>{new ErrorMessage(msg,
otherFields)});
}
```

```
}

else{
saveMonitoringMessagesMap.put(objMonitor, new Map<String, list<ErrorMessage>>{ref => new
list<ErrorMessage>{new ErrorMessage(msg, otherFields)}});
}
}
}

    public static void saveMonitoringMessages(System.Type objMonitor) {
        saveMonitoringMessages(objMonitor, false);
    }

    public static void saveMonitoringMessages() {
        saveMonitoringMessages(null, true);
    }

@TestVisible
    private static void saveMonitoringMessages(System.Type objMonitor, Boolean saveAll) {

        // check the monitoring message map has messages
        if (saveMonitoringMessagesMap != null) {

            Integer parentCount = 0;
            Map<Integer, DebugParent__c> newDbgParents = new Map<Integer, DebugParent__c>();
            Map<Integer, Debugger__c[]> newDbgMessages = new Map<Integer, Debugger__c[]>();

            for (System.Type eachType : saveMonitoringMessagesMap.keySet()) {

                // check if we are to only save one source object type or all
                if (saveAll || objMonitor == eachType) {

                    // get the messages for the source object from the monitoring map
                    Map<String, List<ErrorMessage>> saveMsgs = saveMonitoringMessagesMap.get(eachType);

                    for (String ky : saveMsgs.keySet()) {

                        // create a Debug Parent for each item type
                        newDbgParents.put(parentCount, new DebugParent__c(Run_Category__c = ky));

                        for (ErrorMessage errMsg : saveMsgs.get(ky)) {

                            // create the Debug Message record
                            Debugger__c newDbg = new Debugger__c(msg__c = errMsg.msg);

                            // update additional field values
                            newDbg = (Debugger__c) UtilityObjectData.setSObjectFieldValues((Sobject)newDbg,
errMsg.additionalFields);

                            // make a list of debug messages for this parent
```

```
                    if (!newDbgMessages.containsKey(parentCount)) {
                        newDbgMessages.put(parentCount, new Debugger__c[]{});
                    }

                    // add a new Debug Message record to the list
                    newDbgMessages.get(parentCount).add(newDbg);
                }
                ++parentCount;
            }
        }
    }

    // save Debug Parent and Debug Message records
    insertDebugRecords(newDbgParents, newDbgMessages);

    // remove saved messages from the monitor message map
    if (saveAll) {
        saveMonitoringMessagesMap.clear();
    }
    else {
        saveMonitoringMessagesMap.remove(objMonitor);
    }
  }
}


  @TestVisible private static Boolean insertDebugRecords( Map<Integer, DebugParent__c>
newDbgParents,
      Map<Integer, Debugger__c[]> newDbgMessages ) {

    // Don't let any trappable errors cause the transaction to fail
    try {
      // check we have sufficient governor limits for number of DML statements
      if (Limits.getDmlStatements() <= (Limits.getLimitDmlStatements() -2)) {

        // save debug parent records
        if (!newDbgParents.isEmpty()) {

          // check we have sufficient governor limits (number of DML records) for the Debug Parent
records
          if (newDbgParents.size() + Limits.getDmlRows() < Limits.getLimitDmlRows()) {

            insert newDbgParents.values();

            // create list of Debug Messages with link to correct Debug Parent record
            Debugger__c[] totalDbgMessages = new Debugger__c[]{};
            for (Integer thisParent : newDbgMessages.keySet()) {
              DebugParent__c debugParent = newDbgParents.get(thisParent);
              for (Debugger__c debugMessage : newDbgMessages.get(thisParent)) {
                debugMessage.DebugParent__c = debugParent.Id;
```

```
                        // check we have sufficient governor limits (number of DML records) for the Debug
Message
                        if (totalDbgMessages.size() + Limits.getDmlRows() < Limits.getLimitDmlRows()) {
                            totalDbgMessages.add(debugMessage);
                        }
                    }
                }
                // save debug message records
                if (!totalDbgMessages.isEmpty()) {
                    insert totalDbgMessages;
                }
            }
        }

        return true;

    }
    catch( Exception e ) {
        return false;
    }
  }

}
```

SCRIPT USED TO CREATE NEW JSON STRINGS

```
/*Example of a script to produce Json string of 1 Account record. To create your specific Json string replace
the SOQL highlighted in bold, run the script and copy the Json string from the debug message at the end of
the debug log
*/

String str = JSON.serialize([Select Name From Account where id='<<id of Account>>' limit 1]);

while( str.indexof('"Id"') != -1){
        integer pos1 = str.indexof('"Id"');
```

```
            integer pos2 = str.indexof("",pos1+7);
            str = str.substring(0,pos1) + str.substring(pos2+2,str.length()) ;
            integer pos3 = str.indexof(',"url":');
            integer pos4 = str.indexof('}',pos3);
            str = str.substring(0,pos3) + str.substring(pos4,str.length()) ;
}

str = str.replaceall(',,','},');
str = str.replaceall(',]','}]');

system.debug('Output' + str);
```

EXAMPLES USING THE FRAMEWORK

EXAMPLE 1

```
//Inserts an standard Account with standard json string
TestDataRelatedObjects td = new TestDataRelatedObjects();

        Account acc = td.insertAccount(null,null);
```

EXAMPLE 2

```
//Inserts an Account using standard json string but overriding a field value
TestDataRelatedObjects td = new TestDataRelatedObjects();

        Account acc = td.insertAccount(new Map<String, Object>{'Type__c' => 'xxx'},null);
```

EXAMPLE 3

```
//Inserts an Account and Contact related to each other
TestDataRelatedObjects td = new TestDataRelatedObjects();

        Account acc = td.insertAccountAndContact(null,null,null);
```

EXAMPLE 4

```
//Inserts an Account and Contact related to each other using standard json string but
overriding a field value

TestDataRelatedObjects td = new TestDataRelatedObjects();

        Map<System.Type, Map<String, Object>> kMaps = new Map<System.Type,
Map<String, Object>>{Account.class => new Map<String, Object>{'Type__c' => 'xxx'}};
        Account acc = td.insertAccountAndContact(kMaps,null,null);
```

EXAMPLE 5

```
//Inserts an Account and Opportunity. The Contact is not inserted and instead uses a different
Contact already created. All records are related to each other as per the data model. A field
value on the Account is changed

TestDataRelatedObjects td = new TestDataRelatedObjects();

Contact ct = td.insertContact(null,null);
Map<System.Type,id> replaceObjects = new Map<System.Type,id>{Contact.class => ct.id};

        Map<System.Type, Map<String, Object>> kMaps = new Map<System.Type,
Map<String, Object>>{Account.class => new Map<String, Object>{'Type__c' => 'xxx'}};
        Account acc = td.insertAccountContactAndOpportunity(kMaps, replaceObjects,null);
```