

```

trigger AccountAddressTrigger on Account (before insert, before update) {
    For(Account accountAddress: Trigger.new){
        if(accountAddress.BillingPostalCode !=null &&
accountAddress.Match_Billing_Address__c ==true){
            accountAddress.ShippingPostalCode=accountAddress.BillingPostalCode;
        }
    }
}

```

```

trigger ClosedOpportunityTrigger on Opportunity (after insert, after update) {
    List<Task> newtsk = new List<Task>();
    if(trigger.IsAfter && (trigger.IsInsert || trigger.IsUpdate)){
        for(Opportunity op:Trigger.New){
            if(op.StageName == 'Closed Won'){
                Task tsk = new Task();
                tsk.Subject = 'Follow Up Test Task';
                tsk.WhatId = op.id;
                newtsk.add(tsk);
            }
        }
    }
    if(newtsk.size()>0){

```

```
        insert newtsk;
    }
}
```

```
public class VerifyDate {

    //method to handle potential checks against two dates
    public static Date CheckDates(Date date1, Date date2) {

        //if date2 is within the next 30 days of date1, use date2. Otherwise use the end of the
        month

        if(DateWithin30Days(date1,date2)) {
            return date2;
        } else {
            return SetEndOfMonthDate(date1);
        }
    }

    //method to check if date2 is within the next 30 days of date1
    private static Boolean DateWithin30Days(Date date1, Date date2) {

        //check for date2 being in the past
        if( date2 < date1) { return false; }

        //check that date2 is within (>=) 30 days of date1
        Date date30Days = date1.addDays(30); //create a date 30 days away from date1
        if( date2 >= date30Days ) { return false; }
```

```
else { return true; }  
}
```

```
//method to return the end of the month of a given date  
private static Date SetEndOfMonthDate(Date date1) {  
    Integer totalDays = Date.daysInMonth(date1.year(), date1.month());  
    Date lastDay = Date.newInstance(date1.year(), date1.month(), totalDays);  
    return lastDay;  
}  
  
}
```

```
@isTest  
public class TestVerifyDate {  
    @isTest static void testOldDate(){  
        Date dateTest = VerifyDate.CheckDates(date.today(), date.today().addDays(-1));  
        System.assertEquals(date.newInstance(2022, 4, 31), dateTest);  
    }  
}
```

```
@isTest static void testLessThan30Days(){  
    Date dateTest = VerifyDate.CheckDates(date.today(), date.today().addDays(20));  
    System.assertEquals(date.today().addDays(20), dateTest);  
}
```

```
}
```

```
@isTest static void testMoreThan30Days(){
```

```
    Date dateTest = VerifyDate.CheckDates(date.today(), date.today().addDays(31));
```

```
    System.assertEquals(date.newInstance(2022, 4, 31), dateTest);
```

```
}
```

```
}
```

```
trigger RestrictContactByName on Contact (before insert, before update) {
```

```
//check contacts prior to insert or update for invalid data
```

```
For (Contact c : Trigger.New) {
```

```
if(c.LastName == 'INVALIDNAME') { //invalidname is invalid
```

```
c.AddError('The Last Name "'+c.LastName+'" is not allowed for DML');
```

```
}
```

```
}
```

```
}
```

@isTest

private class TestRestrictContactByName {

@isTest static void testInvalidName() {

//try inserting a Contact with INVALIDNAME

Contact myConact = new Contact(LastName='INVALIDNAME');

insert myConact;

// Perform test

Test.startTest();

Database.SaveResult result = Database.insert(myConact, false);

Test.stopTest();

// Verify

// In this case the creation should have been stopped by the trigger,

// so verify that we got back an error.

System.assert(!result.isSuccess());

System.assert(result.getErrors().size() > 0);

System.assertEquals('Cannot create contact with invalid last name.',

result.getErrors()[0].getMessage());

}

}

```
//@isTest
```

```
public class RandomContactFactory {
```

```
    public static List<Contact> generateRandomContacts(Integer  
numContactsToGenerate, String FName) {
```

```
        List<Contact> contactList = new List<Contact>();
```

```
        for(Integer i=0;i<numContactsToGenerate;i++) {
```

```
            Contact c = new Contact(FirstName=FName + ' ' + i, LastName = 'Contact ' + i);
```

```
            contactList.add(c);
```

```
            System.debug(c);
```

```
        }
```

```
        //insert contactList;
```

```
        System.debug(contactList.size());
```

```
        return contactList;
```

```
    }
```

```
public class AccountProcessor {
```

```
    //Writing the countContacts method and marking it with the @future label.
```

@future

```
public static void countContacts(Set<Id> accountIDs) {
```

```
    // Creating a list that will contain all those accounts that are referenced through the  
    accountIDs list.
```

```
    List<Account> accounts = [SELECT Id, Number_of_Contacts__c, (SELECT id FROM  
    Contacts) from Account where id in :accountIDs];
```

```
    //Assignment from the total contact number to the Number_of_Contacts__c field for  
    each account at accounts list.
```

```
    for( Account account : accounts ) {
```

```
        account.Number_of_Contacts__c = account.contacts.size();
```

```
    }
```

```
    //Updating all accounts in list
```

```
    update accounts;
```

```
}
```

```
}
```

@isTest

```
public class AccountProcessorTest {

    @isTest

    public static void countContactsTest(){

        //Creating an account and inserting it

        Account account = New Account(Name = 'Account Number 1');

        insert account;

        //Creating some contacts related to the account and inserting them

        List<Contact> contacts = new List<Contact>();

        contacts.add(New Contact(lastname = 'Related Contact 1', AccountId =
account.Id));

        contacts.add(New Contact(lastname = 'Related Contact 2', AccountId =
account.Id));

        contacts.add(New Contact(lastname = 'Related Contact 3', AccountId =
account.Id));

        contacts.add(New Contact(lastname = 'Related Contact 4', AccountId =
account.Id));

        insert contacts;

        //Creating a List with account Ids to pass them through the
AccountProcessor.countContacts method

        Set<Id> accountIds = new Set<Id>();

        accountIds.add(account.id);

        //Starting Test:
```



```

Test.startTest();

//Calling the AccountProcessor.countContacts method
AccountProcessor.countContacts(accountIds);

//Finishing Test:
Test.stopTest();

Account ACC = [SELECT Number_of_Contacts__c FROM Account WHERE id =
:account.Id LIMIT 1];

//Setting Assert (We have to parse the account.Number_of_Contacts__c
//to integer to avoid some comparasion error between decimal and integer)
System.assertEquals( Integer.valueOf(ACC.Number_of_Contacts__c) , 4);
}

}

```

global class LeadProcessor implements

```
Database.Batchable<sObject>, Database.Stateful {
```

```

// instance member to retain state across transactions
global Integer recordsProcessed = 0;

global Database.QueryLocator start(Database.BatchableContext bc) {
return Database.getQueryLocator('SELECT Id, LeadSource FROM Lead');
}

global void execute(Database.BatchableContext bc, List<Lead> scope){
// process each batch of records
List<Lead> leads = new List<Lead>();
for (Lead lead : scope) {

lead.LeadSource = 'Dreamforce';
// increment the instance member counter
recordsProcessed = recordsProcessed + 1;

}
update leads;
}

global void finish(Database.BatchableContext bc){
System.debug(recordsProcessed + ' records processed. Shazam!');

}

```

```
}
```

```
@isTest
```

```
public class LeadProcessorTest {
```

```
@testSetup
```

```
static void setup() {
```

```
List<Lead> leads = new List<Lead>();
```

```
// insert 200 leads
```

```
for (Integer i=0;i<200;i++) {
```

```
leads.add(new Lead(LastName='Lead '+i,
```

```
Company='Lead', Status='Open - Not Contacted'));
```

```
}
```

```
insert leads;
```

```
}
```

```
static testmethod void test() {
```

```
Test.startTest();
```

```
LeadProcessor lp = new LeadProcessor();
```

```
Id batchId = Database.executeBatch(lp, 200);
```

```
Test.stopTest();
```

```
// after the testing stops, assert records were updated properly
System.assertEquals(200, [select count() from lead where LeadSource = 'Dreamforce']);
}
}
```

```
public class AddPrimaryContact implements Queueable {
    public contact c;
    public String state;

    public AddPrimaryContact(Contact c, String state) {
        this.c = c;
        this.state = state;
    }

    public void execute(QueueableContext qc) {
        system.debug('this.c = '+this.c+' this.state = '+this.state);

        List<Account> acc_lst = new List<account>([select id, name, BillingState from
account where account.BillingState = :this.state limit 200]);

        List<contact> c_lst = new List<contact>();
        for(account a: acc_lst) {
            contact c = new contact();
            c = this.c.clone(false, false, false, false);
```

```

        c.AccountId = a.Id;
        c_lst.add(c);
    }
    insert c_lst;
}

}

```

@IsTest

```
public class AddPrimaryContactTest {
```

@IsTest

```
public static void testing() {
```

```
    List<account> acc_lst = new List<account>();
```

```
    for (Integer i=0; i<50;i++) {
```

```
        account a = new account(name=string.valueOf(i),billingstate='NY');
```

```
        system.debug('account a = '+a);
```

```
        acc_lst.add(a);
```

```
    }
```

```
    for (Integer i=0; i<50;i++) {
```

```
        account a = new account(name=string.valueOf(50+i),billingstate='CA');
```

```

        system.debug('account a = '+a);
        acc_lst.add(a);
    }
    insert acc_lst;
    Test.startTest();
    contact c = new contact(lastname='alex');
    AddPrimaryContact apc = new AddPrimaryContact(c,'CA');
    system.debug('apc = '+apc);
    System.enqueueJob(apc);
    Test.stopTest();
    List<contact> c_lst = new List<contact>([select id from contact]);
    Integer size = c_lst.size();
    system.assertEquals(50, size);
}

}

```

```

global class DailyLeadProcessor implements Schedulable{
    global void execute(SchedulableContext ctx){
        List<Lead> leads = [SELECT Id, LeadSource FROM Lead WHERE LeadSource = "];
    }
}

```

```

if(leads.size() > 0){
    List<Lead> newLeads = new List<Lead>();

    for(Lead lead : leads){
        lead.LeadSource = 'DreamForce';
        newLeads.add(lead);
    }

    update newLeads;
}
}
}

```

@isTest

```

private class DailyLeadProcessorTest{
    //Seconds Minutes Hours Day_of_month Month Day_of_week optional_year
    public static String CRON_EXP = '0 0 0 2 6 ? 2022';

    static testmethod void testScheduledJob(){
        List<Lead> leads = new List<Lead>();
    }
}

```

```

    for(Integer i = 0; i < 200; i++){
        Lead lead = new Lead(LastName = 'Test ' + i, LeadSource = "", Company = 'Test
Company ' + i, Status = 'Open - Not Contacted');
        leads.add(lead);
    }

    insert leads;

    Test.startTest();

    // Schedule the test job

    String jobId = System.schedule('Update LeadSource to DreamForce', CRON_EXP,
new DailyLeadProcessor());

    // Stopping the test will run the job synchronously
    Test.stopTest();
}
}

```

```

public class AnimalLocator{

    public static String getAnimalNameById(Integer x){

        Http http = new Http();

        HttpRequest req = new HttpRequest();
    }
}

```



```

        req.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals/'
+ x);

        req.setMethod('GET');

        Map<String, Object> animal= new Map<String, Object>();

        HttpResponse res = http.send(req);

        if (res.getStatusCode() == 200) {

            Map<String, Object> results = (Map<String,
Object>)JSON.deserializeUntyped(res.getBody());

            animal = (Map<String, Object>) results.get('animal');

        }

        return (String)animal.get('name');

    }

}

```

@isTest

```

private class AnimalLocatorTest{

    @isTest static void AnimalLocatorMock1() {

        Test.setMock(HttpCalloutMock.class, new AnimalLocatorMock());

        string result = AnimalLocator.getAnimalNameById(3);

        String expectedResult = 'chicken';

        System.assertEquals(result,expectedResult );

    }

}

```

```
}
```

```
@isTest
```

```
global class AnimalLocatorMock implements HttpCalloutMock {
```

```
    // Implement this interface method
```

```
    global HTTPResponse respond(HTTPRequest request) {
```

```
        // Create a fake response
```

```
        HTTPResponse response = new HTTPResponse();
```

```
        response.setHeader('Content-Type', 'application/json');
```

```
        response.setBody('{\"animals\": [\"majestic badger\", \"fluffy bunny\", \"scary bear\",  
\"chicken\", \"mighty moose\"]}');
```

```
        response.setStatusCode(200);
```

```
        return response;
```

```
    }
```

```
}
```

```
//Generated by wsdl2apex
```

```

public class ParkService {
    public class byCountryResponse {
        public String[] return_x;

        private String[] return_x_type_info = new
String[]{"return",'http://parks.services/',null,'0','-1','false'};

        private String[] apex_schema_type_info = new
String[]{"http://parks.services/','false','false'};

        private String[] field_order_type_info = new String[]{"return_x"};
    }

    public class byCountry {
        public String arg0;

        private String[] arg0_type_info = new
String[]{"arg0",'http://parks.services/',null,'0','1','false'};

        private String[] apex_schema_type_info = new
String[]{"http://parks.services/','false','false'};

        private String[] field_order_type_info = new String[]{"arg0"};
    }

    public class ParksImplPort {
        public String endpoint_x = 'https://th-apex-soap-
service.herokuapp.com/service/parks';

        public Map<String,String> inputHttpHeaders_x;

        public Map<String,String> outputHttpHeaders_x;

        public String clientCertName_x;

        public String clientCert_x;

        public String clientCertPasswd_x;

        public Integer timeout_x;
    }
}

```

```

    private String[] ns_map_type_info = new String[]{"http://parks.services/",
'ParkService'};

    public String[] byCountry(String arg0) {

        ParkService.byCountry request_x = new ParkService.byCountry();

        request_x.arg0 = arg0;

        ParkService.byCountryResponse response_x;

        Map<String, ParkService.byCountryResponse> response_map_x = new
Map<String, ParkService.byCountryResponse>();

        response_map_x.put('response_x', response_x);

        WebServiceCallout.invoke(

            this,

            request_x,

            response_map_x,

            new String[]{"endpoint_x",

            ",

            'http://parks.services/',

            'byCountry',

            'http://parks.services/',

            'byCountryResponse',

            'ParkService.byCountryResponse'}

        );

        response_x = response_map_x.get('response_x');

        return response_x.return_x;

    }

}

```

```
}
```

```
@isTest
```

```
global class ParkServiceMock implements WebServiceMock {
```

```
    global void doInvoke(
```

```
        Object stub,
```

```
        Object request,
```

```
        Map<String, Object> response,
```

```
        String endpoint,
```

```
        String soapAction,
```

```
        String requestName,
```

```
        String responseNS,
```

```
        String responseName,
```

```
        String responseType) {
```

```
    // start - specify the response you want to send
```

```
    ParkService.byCountryResponse response_x = new  
    ParkService.byCountryResponse();
```

```
    response_x.return_x = new List<String>{'Yellowstone', 'Mackinac National Park',  
'Yosemite'};
```

```
    // end
```

```
    response.put('response_x', response_x);
```

```
}
```

```
}
```

```
public class ParkLocator {  
    public static string[] country(string theCountry) {  
        ParkService.ParksImplPort parkSvc = new ParkService.ParksImplPort(); // remove  
space  
        return parkSvc.byCountry(theCountry);  
    }  
}
```

```
@isTest  
private class ParkLocatorTest {  
    @isTest static void testCallout() {  
        Test.setMock(WebServiceMock.class, new ParkServiceMock ());  
        String country = 'United States';  
        List<String> result = ParkLocator.country(country);  
        List<String> parks = new List<String>{'Yellowstone', 'Mackinac National Park',  
'Yosemite'};  
        System.assertEquals(parks, result);  
    }  
}
```

```
}
```

```
@RestResource(urlMapping='/Accounts/*/contacts')
```

```
global class AccountManager {
```

```
    @HttpGet
```

```
    global static Account getAccount() {
```

```
        RestRequest req = RestContext.request;
```

```
        String accId = req.requestURI.substringBetween('Accounts/', '/contacts');
```

```
        Account acc = [SELECT Id, Name, (SELECT Id, Name FROM Contacts)
```

```
            FROM Account WHERE Id = :accId];
```

```
        return acc;
```

```
    }
```

```
}
```

```
@isTest
```

```
private class AccountManagerTest {
```

```
    private static testMethod void getAccountTest1() {
```

```

    Id recordId = createTestRecord();

    // Set up a test request

    RestRequest request = new RestRequest();

    request.requestUri = 'https://na1.salesforce.com/services/apexrest/Accounts/'+
recordId + '/contacts' ;

    request.httpMethod = 'GET';

    RestContext.request = request;

    // Call the method to test

    Account thisAccount = AccountManager.getAccount();

    // Verify results

    System.assert(thisAccount != null);

    System.assertEquals('Test record', thisAccount.Name);

}

```

```

// Helper method

static Id createTestRecord() {

    // Create test record

    Account TestAcc = new Account(

        Name='Test record');

    insert TestAcc;

    Contact TestCon= new Contact(

        LastName='Test',

        AccountId = TestAcc.id);
}

```



```
    return TestAcc.Id
```

```
;
```

```
}
```

```
}
```