Create an Apex trigger that sets an account's Shipping Postal Code to match the Billing Postal Code if the Match Billing Address option is selected. Fire the trigger before inserting an account or updating an account.

Pre-Work:

Add a checkbox field to the Account object:

- Field Label: `Match Billing Address`
- Field Name: `Match_Billing_Address`
  Note: The resulting API Name should be `Match_Billing_Address__c`.
- Create an Apex trigger:
  - Name: `AccountAddressTrigger`
  - Object: **Account**
  - Events: before insert and before update
  - Condition: Match Billing Address is true
  - Operation: set the Shipping Postal Code to match the Billing Postal Code

```
trigger AccountAddressTrigger on Account (before insert, before update) {


   for(Account acct:Trigger.new)
   {
      if(acct.Match_Billing_Address__c == True)
         acct.ShippingPostalCode = Acct.BillingPostalCode;
   }
}
```

Create a Bulk Apex trigger

Create a bulkified Apex trigger that adds a follow-up task to an opportunity if its stage is Closed Won. Fire the Apex trigger after inserting or updating an opportunity.

- Create an Apex trigger:
    - Name: `ClosedOpportunityTrigger`
    - Object: **Opportunity**
    - Events: after insert and after update
    - Condition: Stage is `Closed Won`
    - Operation: Create a task:
        - `Subject`: `Follow Up Test Task`
        - `WhatId`: the opportunity ID (associates the task with the opportunity)
    - Bulkify the Apex trigger so that it can insert or update 200 or more opportunities

```
trigger ClosedOpportunityTrigger on Opportunity (after insert, after update) {
        List<Task> taskListToInsert = new List<Task>();
  for(Opportunity oppt:Trigger.new)
  {
    if(oppt.StageName == 'Closed Won')
    {
      task t = new Task();
      t.Subject = 'Follow up Test Task';
      t.WhatId = oppt.Id;
      taskListToInsert.add(t);
    }
```

```
    }
    if(taskListToInsert.size() > 0)
    {
        insert taskListToInsert;
    }
}
```

Create a Unit Test for a Simple Apex Class

Create and install a simple Apex class to test if a date is within a proper range, and if not, returns a date that occurs at the end of the month within the range. You'll copy the code for the class from GitHub. Then write unit tests that achieve 100% code coverage.

- Create an Apex class:
    - Name: `VerifyDate`
    - Code: **Copy from GitHub**
- Place the unit tests in a separate test class:
    - Name: `TestVerifyDate`
    - Goal: 100% code coverage
- Run your test class at least once

```
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
private class TestVerifyDate{
   @isTest static void checkDatesTesttrue(){
      Date date1 = date.today();
```

```
        Date date2 = date1.addDays(29);
        Date t = VerifyDate.CheckDates(date1, date2);
        System.assertEquals(t, date2);
    }
    @isTest static void DateOver(){
        Date date1 = date.today();
        Date date2 = date1.addDays(31);
        Date t = VerifyDate.CheckDates(date1, date2);
        System.assertNotEquals(t, date1);
    }
}
```

Create a Unit Test for a Simple Apex Trigger

Create and install a simple Apex trigger which blocks inserts and updates to any contact with a last name of 'INVALIDNAME'. You'll copy the code for the class from GitHub. Then write unit tests that achieve 100% code coverage.

- Create an Apex trigger on the Contact object
    - Name: `RestrictContactByName`
    - Code: **Copy from GitHub**
- Place the unit tests in a separate test class
    - Name: `TestRestrictContactByName`
    - Goal: 100% test coverage
- Run your test class at least once

```
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() {
                    Contact myConact = new Contact(LastName='INVALIDNAME');
        insert myConact;
        Test.startTest();
        Database.SaveResult result = Database.insert(myConact, false);
        Test.stopTest();
                    System.assert(!result.isSuccess());
        System.assert(result.getErrors().size() > 0);
        System.assertEquals('Cannot create contact with invalid last name.',
                    result.getErrors()[0].getMessage());
    }
}
```

Create a Contact Test Factory

Create an Apex class that returns a list of contacts based on two incoming parameters: the number of contacts to generate and the last name. Do not insert the generated contact records into the database.

NOTE: For the purposes of verifying this hands-on challenge, don't specify the @isTest annotation for either the class or the method, even though it's usually required.

- Create an Apex class in the `public` scope
    - Name: `RandomContactFactory` (without the @isTest annotation)
- Use a Public Static Method to consistently generate contacts with unique first names based on the iterated number in the format Test 1, Test 2 and so on.
    - Method Name: `generateRandomContacts` (without the @isTest annotation)
    - Parameter 1: An integer that controls the number of contacts being generated with unique first names
    - Parameter 2: A string containing the last name of the contacts
    - Return Type: `List < Contact >`

```
public class RandomContactFactory {

    public static List<Contact> generateRandomContacts(Integer genfContacts, String FirstName) {

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


        for(Integer i=0;i<genfContacts;i++) {

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

            contactList.add(c);

            System.debug(c);

        }

        System.debug(contactList.size());

        return contactList;

    }


}
```

Create an Apex class that uses the @future annotation to update Account

records.

Create an Apex class with a future method that accepts a List of Account IDs and updates a custom field on the Account object with the number of contacts associated to the Account. Write unit tests that achieve 100% code coverage for the class. Every hands-on challenge in this module asks you to create a test class.

- Create a field on the Account object:
    - Label: `Number Of Contacts`
    - Name: `Number_Of_Contacts`
    - Type: **Number**
    - This field will hold the total number of Contacts for the Account
- Create an Apex class:
    - Name: `AccountProcessor`
    - Method name: `countContacts`
    - The method must accept a List of Account IDs
    - The method must use the @future annotation
    - The method counts the number of Contact records associated to each Account ID passed to the method and updates the 'Number_Of_Contacts__c' field with this value
- Create an Apex test class:
    - Name: `AccountProcessorTest`
    - The unit tests must cover all lines of code included in the **AccountProcessor** class, resulting in 100% code coverage.
- Before verifying this challenge, run your test class at least once using the Developer Console Run All feature

```
public without sharing class AccountProcessor {



    @future
    public static void countContacts(List<Id> accountIds){
```

```
        List<Account> accounts = [SELECT Id, (SELECT Id FROM Contacts) FROM Account WHERE Id IN
:accountIds];

        for (Account acc: accounts) {
            acc.Number_of_Contacts__c = acc.Contacts.size();
        }
        update accounts;
    }
}

@isTest
private class AccountProcessorTest {

    @isTest
    private static void countContactsTest(){

        List<Account> accounts = new List<Account>();
        for (Integer i=0; i<300; i++) {
            accounts.add(new Account(Name='Test Account' + i));
        }
        insert accounts;

        List<Contact> contacts = new List<Contact>();
        List<Id> accountIds = new List<Id>();
        for (Account acc: accounts) {
            contacts.add(new Contact(FirstName=acc.Name, LastName='TestContact', AccountId=acc.Id));
            accountIds.add(acc.Id);
        }
        insert contacts;
```

```
        // Perform the test
        Test.startTest();
        AccountProcessor.countContacts(accountIds);
        Test.stopTest();

        // Check the result
        List<Account> accs = [SELECT Id, Number_of_Contacts__c FROM Account];
        for (Account acc : accs) {
            System.assertEquals(1, acc.Number_of_Contacts__c, 'ERROR: At least 1 Account record with incorrect Contact count');
        }
    }
}
```

Create an Apex class that uses Batch Apex to update Lead records.

Create an Apex class that implements the Database.Batchable interface to update all Lead records in the org with a specific LeadSource.

- Create an Apex class:
    - Name: `LeadProcessor`
    - Interface: `Database.Batchable`
    - Use a QueryLocator in the start method to collect all Lead records in the org
    - The execute method must update all Lead records in the org with the LeadSource value of `Dreamforce`
- Create an Apex test class:
    - Name: `LeadProcessorTest`

        - In the test class, insert 200 Lead records, execute the LeadProcessor Batch class and test that all Lead records were updated correctly
        - The unit tests must cover all lines of code included in the **LeadProcessor** class, resulting in 100% code coverage
    - Before verifying this challenge, run your test class at least once using the Developer Console Run All feature

```
public without sharing class LeadProcessor implements Database.Batchable<sObject>, Database.Stateful
{


   public Integer recordCount = 0;


   // Collect the Lead records to be passed to execute
   public Database.QueryLocator start(Database.BatchableContext dbc){
      System.debug('Filling the bucket');
      return Database.getQueryLocator([SELECT Id, Name FROM Lead]);
   }


   // Process each batch of records
   public void execute(Database.BatchableContext dbc, List<Lead> leads){
      //System.debug('Job Id ' + dbc.getJobId()); // Returns the batch job ID
      //System.debug('Child Job Id ' + dbc.getChildJobId()); // Returns the ID of the current batch job
chunk that is being processed
      for (Lead l : leads) {
         l.LeadSource = 'Dreamforce';
      }
      update leads;
      recordCount = recordCount + leads.size();
      System.debug('Records processed so far ' + recordCount);
   }
```

```
    // Execute any post-processing operations
    public void finish(Database.BatchableContext dbc){
        System.debug('Total records processed ' + recordCount);
    }
}

@isTest
private class LeadProcessorTest {

    @isTest
    private static void testBatchClass() {

        // Load test data
        List<Lead> leads = new List<Lead>();
        for (Integer i=0; i<200; i++) {
            leads.add(new Lead(LastName='Connock', Company='Salesforce'));
        }
        insert leads;

        // Perform the test
        Test.startTest();
        LeadProcessor lp = new LeadProcessor();
        Id batchId = Database.executeBatch(lp, 200);
        Test.stopTest();

        // Check the result
        List<Lead> updatedLeads = [SELECT Id FROM Lead WHERE LeadSource = 'Dreamforce'];
        System.assertEquals(200, updatedLeads.size(), 'ERROR: At least 1 Lead record not updated
correctly');
```

```
    }
}
```

Create a Queueable Apex class that inserts Contacts for Accounts.

Create a Queueable Apex class that inserts the same Contact for each Account for a specific state.

- Create an Apex class:
    - Name: `AddPrimaryContact`
    - Interface: `Queueable`
    - Create a constructor for the class that accepts as its first argument a Contact sObject and a second argument as a string for the State abbreviation
    - The `execute` method must query for a maximum of 200 Accounts with the `BillingState` specified by the State abbreviation passed into the constructor and insert the Contact sObject record associated to each Account. Look at the sObject `clone()` method.
- Create an Apex test class:
    - Name: `AddPrimaryContactTest`
    - In the test class, insert 50 Account records for `BillingState NY` and 50 Account records for `BillingState CA`
    - Create an instance of the `AddPrimaryContact` class, enqueue the job, and assert that a Contact record was inserted for each of the 50 Accounts with the `BillingState` of `CA`
    - The unit tests must cover all lines of code included in the **AddPrimaryContact** class, resulting in 100% code coverage
- Before verifying this challenge, run your test class at least once using the Developer Console Run All feature

```
public without sharing class AddPrimaryContact implements Queueable {

    private Contact contact;
```

```
    private String state;

        // Constructor - pass in Contact sObject and State abbreviation as arguments
    public AddPrimaryContact(Contact inputContact, String inputState) {

            // Store in class instance variables
        this.contact = inputContact;
        this.state = inputState;
    }

    public void execute(QueueableContext context) {
        //System.debug('Job Id ' + context.getJobId());

            // Retrieve 200 Account records
        List<Account> accounts = [SELECT Id FROM Account WHERE BillingState = :state LIMIT 200];

            // Create empty list of Contact records
        List<Contact> contacts = new List<Contact>();

            // Iterate through the Account records
        for ( Account acc : accounts) {

                // Clone (copy) the Contact record, make the clone a child of the specific
Account record
                // and add to the list of Contacts
            Contact contactClone = contact.clone();
            contactClone.AccountId = acc.Id;
            contacts.add(contactClone);
        }
```

```
                    // Add the new Contact records to the database
        insert contacts;
    }
}



@isTest
private class AddPrimaryContactTest {

    @isTest
    private static void testQueueableClass() {

        // Load test data
        List<Account> accounts = new List<Account>();
        for (Integer i=0; i<500; i++) {
            Account acc = new Account(Name='Test Account');
            if ( i<250 ) {
                acc.BillingState = 'NY';
            } else {
                acc.BillingState = 'CA';
            }
            accounts.add(acc);
        }
        insert accounts;

        Contact contact = new Contact(FirstName='Simon', LastName='Connock');
        insert contact;
```

```
        // Perform the test
        Test.startTest();
        Id jobId = System.enqueueJob(new AddPrimaryContact(contact, 'CA'));
        Test.stopTest();

        // Check the result
        List<Contact> contacts = [SELECT Id FROM Contact WHERE Contact.Account.BillingState =
'CA'];
        System.assertEquals(200, contacts.size(), 'ERROR: Incorrect number of Contact records
found');
    }
}
```

Create an Apex class that uses Scheduled Apex to update Lead records.

Create an Apex class that implements the Schedulable interface to update Lead records with a specific LeadSource. (This is very similar to what you did for Batch Apex.)

- Create an Apex class:
    - Name: `DailyLeadProcessor`
    - Interface: `Schedulable`
    - The execute method must find the first 200 Lead records with a blank LeadSource field and update them with the LeadSource value of `Dreamforce`
- Create an Apex test class:
    - Name: `DailyLeadProcessorTest`
    - In the test class, insert 200 Lead records, schedule the DailyLeadProcessor class to run and test that all Lead records were updated correctly
    - The unit tests must cover all lines of code included in the **DailyLeadProcessor** class, resulting in 100% code coverage.
- Before verifying this challenge, run your test class at least once using the Developer Console Run All feature

```
public without sharing class DailyLeadProcessor implements Schedulable {
```

```
    public void execute(SchedulableContext ctx) {
        //System.debug('Context ' + ctx.getTriggerId()); // Returns the ID of the CronTrigger scheduled job

                        // Get 200 Lead records and modify the LeadSource field
        List<Lead> leads = [SELECT Id, LeadSource FROM Lead WHERE LeadSource = null LIMIT 200];
        for ( Lead l : leads) {
            l.LeadSource = 'Dreamforce';
        }

                        // Update the modified records
        update leads;
    }
}



@isTest
private class DailyLeadProcessorTest {

    private static String CRON_EXP = '0 0 0 ? * * *'; // Midnight every day

    @isTest
    private static void testSchedulableClass() {

        // Load test data
        List<Lead> leads = new List<Lead>();
        for (Integer i=0; i<500; i++) {
            if ( i < 250 ) {
                leads.add(new Lead(LastName='Connock', Company='Salesforce'));
```

```
        } else {
            leads.add(new Lead(LastName='Connock', Company='Salesforce', LeadSource='Other'));
        }
    }
    insert leads;

    // Perform the test
    Test.startTest();
    String jobId = System.schedule('Process Leads', CRON_EXP, new DailyLeadProcessor());
    Test.stopTest();

    // Check the result
    List<Lead> updatedLeads = [SELECT Id, LeadSource FROM Lead WHERE LeadSource = 'Dreamforce'];
    System.assertEquals(200, updatedLeads.size(), 'ERROR: At least 1 record not updated correctly');

    // Check the scheduled time
    List<CronTrigger> cts = [SELECT Id, TimesTriggered, NextFireTime FROM CronTrigger WHERE Id =
:jobId];
    System.debug('Next Fire Time ' + cts[0].NextFireTime);

    // Not sure this works for all timezones
                //Datetime midnight = Datetime.newInstance(Date.today(), Time.newInstance(0,0,0,0));
    //System.assertEquals(midnight.addHours(24), cts[0].NextFireTime, 'ERROR: Not scheduled for
Midnight local time');
  }
}
```

Create an Apex class that calls a REST endpoint and write a test class.

Create an Apex class that calls a REST endpoint to return the name of an animal, write

unit tests that achieve 100% code coverage for the class using a mock response, and run your Apex tests.

Prework: Be sure the Remote Sites from the first unit are set up.

- Create an Apex class:
    - Name: `AnimalLocator`
    - Method name: `getAnimalNameById`
    - The method must accept an Integer and return a String.
    - The method must call https://th-apex-http-callout.herokuapp.com/animals/<id>, replacing <id> with the ID passed into the method
    - The method returns the value of the **name** property (i.e., the animal name)
- Create a test class:
    - Name: `AnimalLocatorTest`
    - The test class uses a mock class called `AnimalLocatorMock` to mock the callout response
- Create unit tests:
    - Unit tests must cover all lines of code included in the **AnimalLocator** class, resulting in 100% code coverage
- Run your test class at least once (via **Run All** tests the Developer Console) before attempting to verify this challenge

```
public class AnimalLocator {


  public static string getAnimalNameById (Integer i) {
    Http http = new Http();
    HttpRequest request = new HttpRequest();
    request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals/'+i);
    request.setMethod('GET');
```

```
        HttpResponse response = http.send(request);


                Map<String, Object> result = (Map<String,
Object>)JSON.deserializeUntyped(response.getBody());
            Map<String, Object> animal = (Map<String, Object>)result.get('animal');
            System.debug('name: '+string.valueOf(animal.get('name')));
            return string.valueOf(animal.get('name'));
    }
}




@isTest
global class AnimalLocatorMock implements HttpCalloutMock {



    global HttpResponse respond(HttpRequest request) {
        HttpResponse response = new HttpResponse();
        response.setHeader('contentType', 'application/json');
        response.setBody('{"animal":{"id":1,"name":"moose","eats":"plants","says":"bellows"}}');
        response.setStatusCode(200);
        return response;
    }
}


____
@isTest
private class AnimalLocatorTest {
```

```
    @isTest
    static void animalLocatorTest() {
        Test.setMock(HttpCalloutMock.class, new AnimalLocatorMock());
        String actual = AnimalLocator.getAnimalNameById(1);
        String expected = 'moose';
        System.assertEquals(actual, expected);
    }
}
```

Generate an Apex class using WSDL2Apex and write a test class.

Generate an Apex class using WSDL2Apex for a SOAP web service, write unit tests that achieve 100% code coverage for the class using a mock response, and run your Apex tests.

Prework: Be sure the Remote Sites from the first unit are set up.

- Generate a class using this using this WSDL file:
    - Name: `ParkService` (Tip: After you click the **Parse WSDL** button, change the Apex class name from **parksServices** to `ParkService`)
    - Class must be in public scope
- Create a class:
    - Name: `ParkLocator`
    - Class must have a **country** method that uses the **ParkService** class
    - Method must return an array of available park names for a particular country passed to the web service (such as Germany, India, Japan, and United States)
- Create a test class:
    - Name: `ParkLocatorTest`
    - Test class uses a mock class called `ParkServiceMock` to mock the callout response

- Create unit tests:
    - Unit tests must cover all lines of code included in the **ParkLocator** class, resulting in 100% code coverage.
- Run your test class at least once (via **Run All** tests the Developer Console) before attempting to verify this challenge.

```
public class ParkLocator {

    public static List<String> country(String country) {
        ParkService.ParksImplPort prkSvc = new ParkService.ParksImplPort();
        return prkSvc.byCountry(country);
    }

}
```

```
@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) {
            parkService.byCountryResponse response_x = new parkService.byCountryResponse();
            response_x.return_x = new List<String>{'Yosemite', 'Sequoia', 'Crater Lake'};
                response.put('response_x', response_x);


        }
}




@isTest
private class ParkLocatorTest {



    @isTest static void testCallout () {
        Test.setMock(WebServiceMock.class, new ParkServiceMock());
        String Country = 'United States';
        List<String> expectedParks = new List<String>{'Yosemite', 'Sequoia', 'Crater Lake'};
            System.assertEquals(expectedParks, ParkLocator.country(country));


    }


}
```

Create an Apex REST service that returns an account and its contacts.
Create an Apex REST class that is accessible at /Accounts/<Account_ID>/contacts. The service will return the account's ID and name plus the ID and name of all contacts associated with the account. Write unit tests that achieve 100% code coverage for the class and run your Apex tests.

Prework: Be sure the Remote Sites from the first unit are set up.

- Create an Apex class
    - Name: `AccountManager`
    - Class must have a method called `getAccount`
    - Method must be annotated with **@HttpGet** and return an **Account** object
    - Method must return the **ID** and **Name** for the requested record and all associated contacts with their **ID** and **Name**
- Create unit tests
    - Unit tests must be in a separate Apex class called `AccountManagerTest`
    - Unit tests must cover all lines of code included in the **AccountManager** class, resulting in 100% code coverage
- Run your test class at least once (via **Run All** tests the Developer Console) before attempting to verify this challenge

```
@RestResource(urlMapping='/Accounts/*/contacts')
global with sharing class AccountManager {


  @HttpGet
  global static Account getAccount() {
    RestRequest request = RestContext.request;
    String accountId = request.requestURI.substringBetween('Accounts/','/contacts');
    Account result = [SELECT ID,Name,(SELECT ID, FirstName, LastName FROM Contacts)
```

```
                FROM Account
                WHERE Id = :accountId];
        return result;
    }

}



@isTest
private class AccountManagerTest {



    @isTest
    static void testGetAccount() {
        Account a = new Account(Name='TestAccount');
        insert a;
        Contact c = new Contact(accountId=a.Id, FirstName='Test', LastName='Test');
        insert c;


        RestRequest request = new RestRequest();
        request.requestUri =
'https://yourInstance.salesfroce.com/services/apexrest/Accounts/'+a.id+'/contacts';
        request.httpMethod = 'GET';
        RestContext.request = request;


        Account myAcct = AccountManager.getAccount();


        System.assert(myAcct != null);
```

```
        System.assertEquals('TestAccount', myAcct.Name);
    }

}
```

APEX SUPERBADGE

Automate Record Creation

```
public with sharing class MaintenanceRequestHelper {
    public static void updateworkOrders(List<Case> updWorkOrders, Map<Id,Case> nonUpdCaseMap) {
        Set<Id> validIds = new Set<Id>();
        For (Case c : updWorkOrders){
            if (nonUpdCaseMap.get(c.Id).Status != 'Closed' && c.Status == 'Closed'){
                if (c.Type == 'Repair' || c.Type == 'Routine Maintenance'){
                    validIds.add(c.Id);
                }
            }
        }

        //When an existing maintenance request of type Repair or Routine Maintenance is closed,
        //create a new maintenance request for a future routine checkup.
        if (!validIds.isEmpty()){
            Map<Id,Case> closedCases = new Map<Id,Case>([SELECT Id, Vehicle__c, Equipment__c,
Equipment__r.Maintenance_Cycle__c,
                                        (SELECT Id,Equipment__c,Quantity__c FROM
Equipment_Maintenance_Items__r)
                                        FROM Case WHERE Id IN :validIds]);
            Map<Id,Decimal> maintenanceCycles = new Map<ID,Decimal>();

            //calculate the maintenance request due dates by using the maintenance cycle defined on the
```

related equipment records.

```
    AggregateResult[] results = [SELECT Maintenance_Request__c,
                                 MIN(Equipment__r.Maintenance_Cycle__c)cycle
                                 FROM Equipment_Maintenance_Item__c
                                 WHERE Maintenance_Request__c IN :ValidIds GROUP BY
Maintenance_Request__c];


    for (AggregateResult ar : results){
       maintenanceCycles.put((Id) ar.get('Maintenance_Request__c'), (Decimal) ar.get('cycle'));
    }


    List<Case> newCases = new List<Case>();
    for(Case cc : closedCases.values()){
       Case nc = new Case (
          ParentId = cc.Id,
          Status = 'New',
          Subject = 'Routine Maintenance',
          Type = 'Routine Maintenance',
          Vehicle__c = cc.Vehicle__c,
          Equipment__c =cc.Equipment__c,
          Origin = 'Web',
          Date_Reported__c = Date.Today()
       );

       //If multiple pieces of equipment are used in the maintenance request,
       //define the due date by applying the shortest maintenance cycle to today's date.
       If (maintenanceCycles.containskey(cc.Id)){
          nc.Date_Due__c = Date.today().addDays((Integer) maintenanceCycles.get(cc.Id));
       } else {
```

```
            nc.Date_Due__c = Date.today().addDays((Integer) cc.Equipment__r.maintenance_Cycle__c);
          }

          newCases.add(nc);
        }

        insert newCases;

        List<Equipment_Maintenance_Item__c> clonedList = new
List<Equipment_Maintenance_Item__c>();
        for (Case nc : newCases){
          for (Equipment_Maintenance_Item__c clonedListItem :
closedCases.get(nc.ParentId).Equipment_Maintenance_Items__r){
            Equipment_Maintenance_Item__c item = clonedListItem.clone();
            item.Maintenance_Request__c = nc.Id;
            clonedList.add(item);
          }
        }
        insert clonedList;
      }
  }
}




trigger MaintenanceRequest on Case (before update, after update) {
  if(Trigger.isUpdate && Trigger.isAfter){
    MaintenanceRequestHelper.updateWorkOrders(Trigger.New, Trigger.OldMap);
  }
```

```
}
```

STEP 3 Synchronize Salesforce

```
public with sharing class WarehouseCalloutService implements Queueable {
   private static final String WAREHOUSE_URL = 'https://th-superbadge-
apex.herokuapp.com/equipment';

   //Write a class that makes a REST callout to an external warehouse system to get a list of
equipment that needs to be updated.
   //The callout's JSON response returns the equipment records that you upsert in Salesforce.

   @future(callout=true)
   public static void runWarehouseEquipmentSync(){
      System.debug('go into runWarehouseEquipmentSync');
      Http http = new Http();
      HttpRequest request = new HttpRequest();

      request.setEndpoint(WAREHOUSE_URL);
      request.setMethod('GET');
      HttpResponse response = http.send(request);

      List<Product2> product2List = new List<Product2>();
      System.debug(response.getStatusCode());
      if (response.getStatusCode() == 200){
         List<Object> jsonResponse =
(List<Object>)JSON.deserializeUntyped(response.getBody());
         System.debug(response.getBody());

         //class maps the following fields:
         //warehouse SKU will be external ID for identifying which equipment records to update
within Salesforce
         for (Object jR : jsonResponse){
            Map<String,Object> mapJson = (Map<String,Object>)jR;
            Product2 product2 = new Product2();
            //replacement part (always true),
            product2.Replacement_Part__c = (Boolean) mapJson.get('replacement');
            //cost
```

```
            product2.Cost__c = (Integer) mapJson.get('cost');
            //current inventory
            product2.Current_Inventory__c = (Double) mapJson.get('quantity');
            //lifespan
            product2.Lifespan_Months__c = (Integer) mapJson.get('lifespan');
            //maintenance cycle
            product2.Maintenance_Cycle__c = (Integer) mapJson.get('maintenanceperiod');
            //warehouse SKU
            product2.Warehouse_SKU__c = (String) mapJson.get('sku');

            product2.Name = (String) mapJson.get('name');
            product2.ProductCode = (String) mapJson.get('_id');
            product2List.add(product2);
        }

        if (product2List.size() > 0){
            upsert product2List;
            System.debug('Your equipment was synced with the warehouse one');
        }
      }
   }

   public static void execute (QueueableContext context){
      System.debug('start runWarehouseEquipmentSync');
      runWarehouseEquipmentSync();
      System.debug('end runWarehouseEquipmentSync');
   }

}
```

STEP 4: Schedule Sync.

```
global with sharing class WarehouseSyncSchedule implements Schedulable{
   global void execute(SchedulableContext ctx){
      System.enqueueJob(new WarehouseCalloutService());
   }
}
```

STEP 5 : Test Automation

```
@isTest
public with sharing class MaintenanceRequestHelperTest {

   // createVehicle
   private static Vehicle__c createVehicle(){
      Vehicle__c vehicle = new Vehicle__C(name = 'Testing Vehicle');
      return vehicle;
   }

   // createEquipment
   private static Product2 createEquipment(){
      product2 equipment = new product2(name = 'Testing equipment',
                                 lifespan_months__c = 10,
                                 maintenance_cycle__c = 10,
                                 replacement_part__c = true);
      return equipment;
   }

   // createMaintenanceRequest
   private static Case createMaintenanceRequest(id vehicleId, id equipmentId){
      case cse = new case(Type='Repair',
                  Status='New',
                  Origin='Web',
                  Subject='Testing subject',
                  Equipment__c=equipmentId,
                  Vehicle__c=vehicleId);
      return cse;
   }

   // createEquipmentMaintenanceItem
   private static Equipment_Maintenance_Item__c createEquipmentMaintenanceItem(id
equipmentId,id requestId){
      Equipment_Maintenance_Item__c equipmentMaintenanceItem = new
Equipment_Maintenance_Item__c(
         Equipment__c = equipmentId,
         Maintenance_Request__c = requestId);
      return equipmentMaintenanceItem;
   }
```

```
    @isTest
    private static void testPositive(){
        Vehicle__c vehicle = createVehicle();
        insert vehicle;
        id vehicleId = vehicle.Id;

        Product2 equipment = createEquipment();
        insert equipment;
        id equipmentId = equipment.Id;

        case createdCase = createMaintenanceRequest(vehicleId,equipmentId);
        insert createdCase;

        Equipment_Maintenance_Item__c equipmentMaintenanceItem =
createEquipmentMaintenanceItem(equipmentId,createdCase.id);
        insert equipmentMaintenanceItem;

        test.startTest();
        createdCase.status = 'Closed';
        update createdCase;
        test.stopTest();

        Case newCase = [Select id,
                        subject,
                        type,
                        Equipment__c,
                        Date_Reported__c,
                        Vehicle__c,
                        Date_Due__c
                       from case
                       where status ='New'];

        Equipment_Maintenance_Item__c workPart = [select id
                                    from Equipment_Maintenance_Item__c
                                    where Maintenance_Request__c =:newCase.Id];
        list<case> allCase = [select id from case];
        system.assert(allCase.size() == 2);

        system.assert(newCase != null);
        system.assert(newCase.Subject != null);
        system.assertEquals(newCase.Type, 'Routine Maintenance');
```

```
        SYSTEM.assertEquals(newCase.Equipment__c, equipmentId);
        SYSTEM.assertEquals(newCase.Vehicle__c, vehicleId);
        SYSTEM.assertEquals(newCase.Date_Reported__c, system.today());
    }

    @isTest
    private static void testNegative(){
        Vehicle__C vehicle = createVehicle();
        insert vehicle;
        id vehicleId = vehicle.Id;

        product2 equipment = createEquipment();
        insert equipment;
        id equipmentId = equipment.Id;

        case createdCase = createMaintenanceRequest(vehicleId,equipmentId);
        insert createdCase;

        Equipment_Maintenance_Item__c workP = createEquipmentMaintenanceItem(equipmentId,
createdCase.Id);
        insert workP;

        test.startTest();
        createdCase.Status = 'Working';
        update createdCase;
        test.stopTest();

        list<case> allCase = [select id from case];

        Equipment_Maintenance_Item__c equipmentMaintenanceItem = [select id
                                    from Equipment_Maintenance_Item__c
                                    where Maintenance_Request__c = :createdCase.Id];

        system.assert(equipmentMaintenanceItem != null);
        system.assert(allCase.size() == 1);
    }

    @isTest
    private static void testBulk(){
        list<Vehicle__C> vehicleList = new list<Vehicle__C>();
        list<Product2> equipmentList = new list<Product2>();
```

```
        list<Equipment_Maintenance_Item__c> equipmentMaintenanceItemList = new
list<Equipment_Maintenance_Item__c>();
        list<case> caseList = new list<case>();
        list<id> oldCaseIds = new list<id>();

        for(integer i = 0; i < 300; i++){
            vehicleList.add(createVehicle());
            equipmentList.add(createEquipment());
        }
        insert vehicleList;
        insert equipmentList;

        for(integer i = 0; i < 300; i++){
            caseList.add(createMaintenanceRequest(vehicleList.get(i).id, equipmentList.get(i).id));
        }
        insert caseList;

        for(integer i = 0; i < 300; i++){

equipmentMaintenanceItemList.add(createEquipmentMaintenanceItem(equipmentList.get(i).id,
caseList.get(i).id));
        }
        insert equipmentMaintenanceItemList;

        test.startTest();
        for(case cs : caseList){
            cs.Status = 'Closed';
            oldCaseIds.add(cs.Id);
        }
        update caseList;
        test.stopTest();

        list<case> newCase = [select id
                              from case
                              where status ='New'];



        list<Equipment_Maintenance_Item__c> workParts = [select id
                                                         from Equipment_Maintenance_Item__c
                                                         where Maintenance_Request__c in: oldCaseIds];
```

```
        system.assert(newCase.size() == 300);

        list<case> allCase = [select id from case];
        system.assert(allCase.size() == 600);
    }
}
```

STEP 6: TEST Callout

```
@IsTest
private class WarehouseCalloutServiceTest {
    // implement your mock callout test here
            @isTest
    static void testWarehouseCallout() {
        test.startTest();
        test.setMock(HttpCalloutMock.class, new WarehouseCalloutServiceMock());
        WarehouseCalloutService.execute(null);
        test.stopTest();

        List<Product2> product2List = new List<Product2>();
        product2List = [SELECT ProductCode FROM Product2];

        System.assertEquals(3, product2List.size());
        System.assertEquals('55d66226726b611100aaf741', product2List.get(0).ProductCode);
        System.assertEquals('55d66226726b611100aaf742', product2List.get(1).ProductCode);
        System.assertEquals('55d66226726b611100aaf743', product2List.get(2).ProductCode);
    }
}
```

```
@isTest
global class WarehouseCalloutServiceMock implements HttpCalloutMock {
    // implement http mock callout
    global static HttpResponse respond(HttpRequest request) {

        HttpResponse response = new HttpResponse();
        response.setHeader('Content-Type', 'application/json');

response.setBody('[{"_id":"55d66226726b611100aaf741","replacement":false,"quantity":5,"name":
```

```
"Generator 1000
kW","maintenanceperiod":365,"lifespan":120,"cost":5000,"sku":"100003"},{"_id":"55d66226726b611
100aaf742","replacement":true,"quantity":183,"name":"Cooling
Fan","maintenanceperiod":0,"lifespan":0,"cost":300,"sku":"100004"},{"_id":"55d66226726b611100a
af743","replacement":true,"quantity":143,"name":"Fuse
20A","maintenanceperiod":0,"lifespan":0,"cost":22,"sku":"100005"}]');
        response.setStatusCode(200);

        return response;
    }
}


public with sharing class WarehouseCalloutService implements Queueable {
    private static final String WAREHOUSE_URL = 'https://th-superbadge-
apex.herokuapp.com/equipment';

    //Write a class that makes a REST callout to an external warehouse system to get a list of
equipment that needs to be updated.
    //The callout's JSON response returns the equipment records that you upsert in Salesforce.

    @future(callout=true)
    public static void runWarehouseEquipmentSync(){
        System.debug('go into runWarehouseEquipmentSync');
        Http http = new Http();
        HttpRequest request = new HttpRequest();

        request.setEndpoint(WAREHOUSE_URL);
        request.setMethod('GET');
        HttpResponse response = http.send(request);

        List<Product2> product2List = new List<Product2>();
        System.debug(response.getStatusCode());
        if (response.getStatusCode() == 200){
            List<Object> jsonResponse =
(List<Object>)JSON.deserializeUntyped(response.getBody());
            System.debug(response.getBody());

            //class maps the following fields:
            //warehouse SKU will be external ID for identifying which equipment records to update
within Salesforce
```

```
            for (Object jR : jsonResponse){
                Map<String,Object> mapJson = (Map<String,Object>)jR;
                Product2 product2 = new Product2();
                //replacement part (always true),
                product2.Replacement_Part__c = (Boolean) mapJson.get('replacement');
                //cost
                product2.Cost__c = (Integer) mapJson.get('cost');
                //current inventory
                product2.Current_Inventory__c = (Double) mapJson.get('quantity');
                //lifespan
                product2.Lifespan_Months__c = (Integer) mapJson.get('lifespan');
                //maintenance cycle
                product2.Maintenance_Cycle__c = (Integer) mapJson.get('maintenanceperiod');
                //warehouse SKU
                product2.Warehouse_SKU__c = (String) mapJson.get('sku');

                product2.Name = (String) mapJson.get('name');
                product2.ProductCode = (String) mapJson.get('_id');
                product2List.add(product2);
            }

            if (product2List.size() > 0){
                upsert product2List;
                System.debug('Your equipment was synced with the warehouse one');
            }
        }
    }

    public static void execute (QueueableContext context){
        System.debug('start runWarehouseEquipmentSync');
        runWarehouseEquipmentSync();
        System.debug('end runWarehouseEquipmentSync');
    }

}
```

STEP 7: Testing Logic

```
@isTest
public with sharing class WarehouseSyncScheduleTest {
    // implement scheduled code here
```

```
    //
    @isTest static void test() {
        String scheduleTime = '00 00 00 * * ? *';
        Test.startTest();
        Test.setMock(HttpCalloutMock.class, new WarehouseCalloutServiceMock());
        String jobId = System.schedule('Warehouse Time to Schedule to test', scheduleTime, new
WarehouseSyncSchedule());
        CronTrigger c = [SELECT State FROM CronTrigger WHERE Id =: jobId];
        System.assertEquals('WAITING', String.valueOf(c.State), 'JobId does not match');

        Test.stopTest();
    }
}


global with sharing class WarehouseSyncSchedule implements Schedulable {
    // implement scheduled code here
    global void execute (SchedulableContext ctx){
        System.enqueueJob(new WarehouseCalloutService());
    }
}


@isTest
global class WarehouseCalloutServiceMock implements HttpCalloutMock {
    // implement http mock callout
    global static HttpResponse respond(HttpRequest request) {

        HttpResponse response = new HttpResponse();
        response.setHeader('Content-Type', 'application/json');

response.setBody('[{"_id":"55d66226726b611100aaf741","replacement":false,"quantity":5,"name":
"Generator 1000
kW","maintenanceperiod":365,"lifespan":120,"cost":5000,"sku":"100003"},{"_id":"55d66226726b611
100aaf742","replacement":true,"quantity":183,"name":"Cooling
Fan","maintenanceperiod":0,"lifespan":0,"cost":300,"sku":"100004"},{"_id":"55d66226726b611100a
af743","replacement":true,"quantity":143,"name":"Fuse
20A","maintenanceperiod":0,"lifespan":0,"cost":22,"sku":"100005"}]');
        response.setStatusCode(200);

        return response;
```

```
    }
}
```