Apex Trigger----------

1.Create an Apex trigger

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

code:

```
trigger AccountAddressTrigger on Account (before insert) {
        for(Account a:Trigger.New){
        If (a.Match_Billing_Address__c && a.BillingPostalCode!= Null) {
           a.BillingPostalCode= a.ShippingPostalCode ;
      }
   }
}
```

```
trigger AccountAddressTrigger on Account (before insert) {
        for(Account a:Trigger.New){
        If (a.Match_Billing_Address__c ==True) {
           a.ShippingPostalCode= a.BillingPostalCode ;
      }
```

```
    }

}
```

2.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

code:

```
trigger ClosedOpportunityTrigger on Opportunity (before insert) {

    List<Task> taskList = new List <task>();



    for(Opportunity opp : Trigger.New){

        if(opp.StageName == 'Closed Won'){

            taskList.add(new Task(Subject = 'Follow Up Test Task', WhatId = opp.Id));

        }

       }

    if(taskList.size()>0){

        insert taskList;

    }



}
```

Apex Testing----------

3.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

code:

VerfyDate class:

```
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;
        }

}
```

TestVerifyDate class:

```
@isTest

public class TestVerifyDate {

    @isTest static void test1(){

        Date d =
VerifyDate.CheckDates(Date.parse('01/01/2022'),Date.parse('01/03/2022'));

        System.assertEquals(Date.parse('01/03/2022'),d);

    }

    @isTest static void test2(){

        Date d =
VerifyDate.CheckDates(Date.parse('01/01/2022'),Date.parse('02/03/2022'));

        System.assertEquals(Date.parse('01/31/2022'),d);

    }

}
```

4.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

code:

trigger name :RestrictContactByName

```
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');

                }



        }



}

class name:TestRestrictContactByName

@isTest

public class TestRestrictContactByName {

   @isTest public static void testContact(){

      Contact ct= new Contact();

      ct.Lastname='INVALIDNAME';

      Database.SaveResult res=Database.insert(ct,false);

      System.assertEquals('The Last Name "INVALIDNAME" is not allowed for DML',
res.getErrors()[0].getMessage());
```

```
    }

}
```

5.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 >

code:

trigger name:RandomContactFactory

```
public class RandomContactFactory {
    public static List<Contact> generateRandomContacts(Integer num,String lastName){
        List<Contact> contactList=new List<Contact>();
      for(Integer i=1;i<=num;i++){
        Contact ct=new Contact(FirstName='Test'+i,LastName=lastName);
        contactList.add(ct);
      }
      return contactList;
   }
}
```

Asynchronous Apex----------

6.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

code:

```
public class AccountProcessor

{

  @future

  public static void countContacts(Set<id> setId)

  {

     List<Account> lstAccount = [select id,Number_of_Contacts__c , (select id from contacts ) from account where id in :setId ];

     for( Account acc : lstAccount )

     {

        List<Contact> lstCont = acc.contacts ;
```

```
        acc.Number_of_Contacts__c = lstCont.size();
      }
      update lstAccount;
  }
}

@IsTest
public class AccountProcessorTest {
    public static testmethod void TestAccountProcessorTest()
    {
        Account a = new Account();
        a.Name = 'Test Account';
        Insert a;

        Contact cont = New Contact();
        cont.FirstName ='Bob';
```

```
        cont.LastName ='Masters';

        cont.AccountId = a.Id;

        Insert cont;



        set<Id> setAccId = new Set<ID>();

        setAccId.add(a.id);



        Test.startTest();

           AccountProcessor.countContacts(setAccId);

        Test.stopTest();



        Account ACC = [select Number_of_Contacts__c from Account where id = :a.id LIMIT
1];

        System.assertEquals ( Integer.valueOf(ACC.Number_of_Contacts__c) ,1);

    }



}
```

7.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

CODE:

```
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']);

    }

}
```

8.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

CODE:

```
public class AddPrimaryContact implements Queueable{

    Contact con;

    String state;



    public AddPrimaryContact(Contact con, String state){

        this.con = con;
```

```
        this.state = state;

    }

    public void execute(QueueableContext qc){

        List<Account> lstOfAccs = [SELECT Id FROM Account WHERE BillingState = :state
LIMIT 200];



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

        for(Account acc : lstOfAccs){

            Contact conInst = con.clone(false,false,false,false);

            conInst.AccountId = acc.Id;



            lstOfConts.add(conInst);

        }



        INSERT lstOfConts;

    }

}
```

```
@isTest
public class AddPrimaryContactTest{
    @testSetup
    static void setup(){
        List<Account> lstOfAcc = new List<Account>();
        for(Integer i = 1; i <= 100; i++){
            if(i <= 50)
                lstOfAcc.add(new Account(name='AC'+i, BillingState = 'NY'));
            else
                lstOfAcc.add(new Account(name='AC'+i, BillingState = 'CA'));
        }

        INSERT lstOfAcc;
    }
```

```
    static testmethod void testAddPrimaryContact(){

        Contact con = new Contact(LastName = 'TestCont');

        AddPrimaryContact addPCIns = new AddPrimaryContact(CON ,'CA');



        Test.startTest();

        System.enqueueJob(addPCIns);

        Test.stopTest();



        System.assertEquals(50, [select count() from Contact]);

    }

}
```

9.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

code:

```
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();

    }

}
```

10.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

code:

```
public class AnimalLocator

{



  public static String getAnimalNameById(Integer id)

   {
```

```
Http http = new Http();

HttpRequest request = new HttpRequest();

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

request.setMethod('GET');

HttpResponse response = http.send(request);

  String strResp = '';

   system.debug('******response '+response.getStatusCode());

   system.debug('******response '+response.getBody());

// If the request is successful, parse the JSON response.

if (response.getStatusCode() == 200)

{

    // Deserializes the JSON string into collections of primitive data types.

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

    // Cast the values in the 'animals' key as a list

   Map<string,object> animals = (map<string,object>) results.get('animal');

    System.debug('Received the following animals:' + animals );
```

```
            strResp = string.valueof(animals.get('name'));
            System.debug('strResp >>>>>>' + strResp );
        }
        return strResp ;
    }

}

@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 {

    global HTTPResponse respond(HTTPRequest request) {

        HttpResponse response = new HttpResponse();

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

        response.setBody('{"animal":{"id":1,"name":"chicken","eats":"chicken
food","says":"cluck cluck"}}');

        response.setStatusCode(200);

        return response;

    }

}
```

12.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.

code:

```
public class ParkLocator {

    public static String[] country(String country){

        ParkService.ParksImplPort parks = new ParkService.ParksImplPort();

        String[] parksname = parks.byCountry(country);

        return parksname;

    }

}
```

```
@isTest

private class ParkLocatorTest{
```

```
    @isTest
    static void testParkLocator() {
        Test.setMock(WebServiceMock.class, new ParkServiceMock());
        String[] arrayOfParks = ParkLocator.country('India');

        System.assertEquals('Park1', arrayOfParks[0]);
    }
}

@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();

    List<String> lstOfDummyParks = new List<String> {'Park1','Park2','Park3'};

    response_x.return_x = lstOfDummyParks;



    response.put('response_x', response_x);

  }

}
```

13.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

code:

```
@RestResource(urlMapping='/Accounts/*/contacts')
global with sharing 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{
    @isTest static void testAccountManager(){
        Id recordId = getTestAccountId();
        // Set up a test request
        RestRequest request = new RestRequest();
        request.requestUri =
            'https://ap5.salesforce.com/services/apexrest/Accounts/'+ recordId +'/contacts';
        request.httpMethod = 'GET';
        RestContext.request = request;

        // Call the method to test
        Account  acc = AccountManager.getAccount();

        // Verify results
```

```
        System.assert(acc != null);
    }

    private static Id getTestAccountId(){
        Account acc = new Account(Name = 'TestAcc2');
        Insert acc;

        Contact con = new Contact(LastName = 'TestCont2', AccountId = acc.Id);
        Insert con;

        return acc.Id;
    }
}
```

Apex Specialist Superbadge:

Challenge2:

MaintenanceRequest.apxt:

```
trigger MaintenanceRequest on Case (before update, after update) {
    //ToDo: Call MaintenanceRequestHelper.updateWorkOrders
    if(trigger.isAfter){
        MaintenanceRequestHelper.updateWorkOrders();
    }
}
```

MaintenanceRequestHelper.apxc

```
public with sharing class MaintenanceRequestHelper {
    public static void updateWorkOrders() {
        List<case> newCaseList = new List<case>();
        Integer avgAmount=10000;

        List<Equipment_Maintenance_Item__c> newEMI = new
```

```
List<Equipment_Maintenance_Item__c>();

    List<case> caseList = [SELECT id,Vehicle__c,Subject,ProductID,Product__c, (SELECT id from Equipment_Maintenance_Items__r) from case where status='closed' and Type IN ('Repair', 'Routine Maintenance') and ID IN :Trigger.new LIMIT 200];

    Map<id,Equipment_Maintenance_Item__c> equip = new map<id,Equipment_Maintenance_Item__c>([Select ID, Equipment__c, Quantity__c,Equipment__r.id,Equipment__r.Maintenance_Cycle__c from Equipment_Maintenance_Item__c ]);

    for(case c: caseList){

        case newCase = new Case();

        newCase.Type = 'Routine Maintenance';

        newCase.Status = 'New';

        newCase.Vehicle__c = c.Vehicle__c;

        newCase.Subject =  String.isBlank(c.Subject) ? 'Routine Maintenance Request' : c.Subject;

        newCase.Date_Reported__c = Date.today();

        newCase.ProductId = c.ProductId;

        newCase.Product__c = c.Product__c;

        newCase.parentID = c.Id;
```

```
        for(Equipment_Maintenance_Item__c emi : c.Equipment_Maintenance_Items__r ){

            avgAmount =
Math.min(avgAmount,Integer.valueOf(equip.get(emi.id).Equipment__r.Maintenance_Cycl
e__c));

            newEMI.add(new Equipment_Maintenance_Item__c(

                Equipment__c = equip.get(emi.id).Equipment__c,

                Maintenance_Request__c = c.id,

                Quantity__c = equip.get(emi.id).Quantity__c));

        }

        Date dueDate = date.TODAY().adddays(avgAmount);

        newCase.Date_Due__c =dueDate;

        newCaseList.add(newCase);



    }

    if(newCaseList.size()>0){

        Database.insert(newCaseList);
```

```
        }


        for(Case c2: newCaseList){
            for(Equipment_Maintenance_Item__c emi2 : newEmi){
                if(c2.parentID == emi2.Maintenance_Request__c){
                    emi2.Maintenance_Request__c = c2.id;
                }
            }
        }


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

Challenge3:

WarehouseCalloutService.apxc:

```
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');
    }

}
```

Debug: system.enqueueJob(WarehouseCalloutService());

Runall

Challenge4:

WarehouseSyncSchedule.apxc:

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


    }
}
```

Challenge5:

MaintenanceRequestHelperTest.apxc:

```
@istest
public with sharing class MaintenanceRequestHelperTest {
    @istest
    public static void BulkTesting(){
        product2 pt2 = new product2(Name = 'tester',Maintenance_Cycle__c = 10,
Replacement_Part__c = true);



        Database.insert(pt2);




        List<case> caseList = new List<case>();
        for(Integer i=0;i<300;i++){
            caseList.add(new case(
                Type = 'Routine Maintenance',
```

```
            Status = 'Closed',

            Subject = 'testing',

            Date_Reported__c = Date.today(),

            ProductId = pt2.id

        ));

    }

    if(caseList.size()>0){

        Database.insert(caseList);

        System.debug(pt2.id);

        System.debug(caseList.size());

    }




    List<Equipment_Maintenance_Item__c> newEMI = new
List<Equipment_Maintenance_Item__c>();

    for(Integer i=0;i<5;i++){

        newEMI.add(new Equipment_Maintenance_Item__c(
```

```
        Equipment__c = pt2.id,
        Maintenance_Request__c = caseList[1].id,
        Quantity__c = 10));
    }
    if(newEmi.size()>0){
       Database.insert(newEmi);
    }

    for(case c :caseList){
       c.Subject = 'For Testing';
    }
    Database.update(caseList);
    Integer newcase = [Select count() from case where ParentId = :caseList[0].id];
    System.assertEquals(1, newcase);

}
```

```
@istest

public static void positive(){

    product2 pt2 = new product2(Name = 'tester',Maintenance_Cycle__c = 10);

    insert pt2;



    Case cParent = new Case(Type = 'Repair',status = 'Closed',Date_Reported__c =
Date.today(),

                    ProductId = pt2.id);

    insert cParent;

    Case cChild = new Case(Type = 'Repair',status = 'Closed',Date_Reported__c =
Date.today(),

                   ProductId = pt2.id,parentID = cParent.ParentId);

    insert cChild;



    cParent.subject = 'child refrecer record';

    update cParent;
```

```
    Integer newcase = [Select count() from case where ParentId = :cParent.id];
    System.assertEquals(1, newcase);

}
@istest public static void negetive(){
    product2 pt2 = new product2(Name = 'tester',Maintenance_Cycle__c = 10);
    insert pt2;

    Case c = new Case(Type = 'Repair',status = 'New',Date_Reported__c = Date.today(),
                ProductId = pt2.id);
    insert c;

    c.Status = 'Working';
    update c;
```

```
        Integer newcase = [Select count() from case where ParentId = :c.id];

        System.assertEquals(0, newcase);

    }

}
```

Challenge6:

WarehouseCalloutServiceMock.apxc:

```
@istest

global class WarehouseCalloutServiceMock implements HttpCalloutMock{

    // implement http mock callout

    global HttpResponse respond(HttpRequest request){

        HttpResponse response = new HttpResponse();

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


response.setBody('[{"_id":"55d66226726b611100aaf741","replacement":true,"quantity":5,"
name":"Generator 1000
kW","maintenanceperiod":365,"lifespan":120,"cost":5000,"sku":"220000"}]');

        response.setStatusCode(200);

        return response;

    }



}
```

WarehouseCalloutServiceTest.apxc:

```
@IsTest
private class WarehouseCalloutServiceTest {
    // implement your mock callout test here
    @isTest static void mainTest(){
        Test.setMock(HttpCalloutMock.class, new WarehouseCalloutServiceMock());
        Test.startTest();
        Id jobID = System.enqueueJob(new WarehouseCalloutService());
        //System.assertEquals('Queued',aaj.status);
        Test.stopTest();
        AsyncApexJob aaj = [SELECT Id, Status, NumberOfErrors FROM AsyncApexJob
WHERE Id = :jobID];
        System.assertEquals('Completed',aaj.status);
        System.assertEquals(0, aaj.NumberOfErrors);
    }
}
```

Challenge7:

WarehouseSyncScheduleTest.apxc

```
@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();
```

```
    }

}
```