Apex Specialist

Apex Triggers

CHALLENGE : Get Started with Apex Triggers

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

Solution:

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

    for(Account account:Trigger.New){
        if(account.Match_Billing_Address__c == True){
            account.ShippingPostalCode =
   account.BillingPostalCode;
        }
    }
}
```

Apex Specialist

CHALLENGE : Bulk Apex Triggers

Create a bullied 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

Solution:

```
trigger ClosedOpportunityTrigger on Opportunity (after
   insert,after update) {
    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 Specialist

Apex Testing

CHALLENGE : Get Started with Apex Unit Tests

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

Solution:

VerifyDate.apxc

```
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.apxc

```
@IsTest
public class TestVerifyDate{
    @isTest static void date2within30daydate1(){
```

```
        Date
    returnDate1=VerifyDate.CheckDates(date.valueOf('2022-05-
    19'),date.valueOf('2022-05-27'));
        System.assertEquals(date.valueOf('2022-05-
    19'),returnDate1);
     }
     @isTest static void date2NOTwithin30daydate1(){
        Date
    returnDate2=VerifyDate.CheckDates(date.valueOf('2022-05-
    19'),date.valueOf('2022-06-24'));
        System.assertEquals(date.valueOf('2022-05-
    31'),returnDate2);
     }
}
```

Apex Specialist

CHALLENGE : Test Apex Triggers

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

Solution:

`RestrictContactByName.apxt`

```
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');
    }

  }
}
```

`TestRestrictContactByName.apxc`

```
@IsTest
public class TestRestrictContactByName {
 @isTest static void createBadContact() {
        Contact c = new
   Contact(FirstName='John',LastName='INVALIDNAME');
        Test.startTest();
        Database.SaveResult result = Database.insert(c, false);
        Test.stopTest();
        System.assert(!result.isSuccess());
    }
}
```

CHALLENGE : Create Test Data for Apex Tests

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 >`

Solution :

```
public class RandomContactFactory {
    public static List<Contact> generateRandomContacts(Integer
   numcnt,string lastname){
        List<Contact> contacts = new List<Contact>();
        for(Integer i=0;i<numcnt;i++){
            Contact cnt = new Contact(FirstName =
   'Test'+i,LastName = Lastname);
```

```
            contacts.add(cnt);
        }
        return contacts;
    }
}
```

Apex Specialist

Asynchronous Apex

CHALLENGE : Use Future Methods

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

Solution :

`AccountProcessor.apxc`

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

```
        List<Account> accountsToUpdate = new
  List<Account>();
  List<Account> accounts = [Select Id, Name,(Select Id from Contacts) from Account
  Where Id IN :accountIds];
          For(Account acc:accounts){
              List<Contact> contactList = acc.Contacts;
              acc.Number_of_Contacts__c=contactList.size()
  ;
              accountsToUpdate.add(acc);
          }
          update accountsToUpdate;
  }
}
```

AccountProcessorTest.apxc

```
@IsTest
private class AccountProcessorTest {
  @IsTest
  private static void testCountContacts() {
 Account newAccount = new Account(Name='Test Account');
    insert newAccount;

    Contact newContact1 = new Contact(FirstName='John',
                                      LastName='Doe',
                                      AccountId=newAccount.
  Id);
    insert newContact1;

    Contact newContact2 = new Contact(FirstName='John',
                                      LastName='Doe',
                                      AccountId=newAccount.
  Id);

    insert newContact2;
    List<Id> accountIds = new List<Id>();
 accountIds.add(newAccount.Id);
    Test.startTest();

 AccountProcessor.countContacts(accountIds);
    Test.stopTest();
  }
}
```

CHALLENGE : Use Batch Apex

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

Solution :

`LeadProcessor.apxc`

```
public class LeadProcessor implements
    Database.Batchable<sObject> {
    public Database.QueryLocator start(Database.BatchableContext
   bc) {
        return Database.getQueryLocator(
            'SELECT ID from Lead'
        );
    }
    public 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';
            leads.add(lead);
            }
        update leads;
    }
    public void finish(Database.BatchableContext bc){

    }
}
```

`LeadProcessorTest.apxc`

```
@isTest
private class LeadProcessorTest {
    @testSetup
    static void setup() {
        List<Lead> leads = new List<Lead>();
        // insert 10 accounts
        for (Integer i=0;i<200;i++) {
            leads.add(new Lead(LastName='Lead
  '+i,Company='Test Co'));
        }
        insert leads;
    }
    @isTest static void test() {
        Test.startTest();
        LeadProcessor myleads = new LeadProcessor();
  Id batchId = Database.executeBatch(myleads);
        Test.stopTest();
        // after the testing stops, assert records were
  updated properly
        System.assertEquals(200, [select count() from Lead
  where LeadSource = 'Dreamforce']);
    }
}
```

CHALLENGE : Control Processes with Queueable Apex

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

Solution :

`AddPrimaryContact.apxc`

```
public class AddPrimaryContact implements Queueable {

    private Contact con;
    private String state;

    public AddPrimaryContact(Contact con, String state) {
        this.con = con;
        this.state = state;
    }
    public void execute(QueueableContext context) {
  List<Account> accounts = [Select Id, Name, (Select
   FirstName,LastName,Id from contacts)
                                  from Account where
   Billingstate =:state Limit 200];
        List<Contact> primaryContacts = new
   List<Contact>();
        for(Account acc:accounts){
            Contact c= con.clone();
            c.AccountId=acc.Id;
            primaryContacts.add(c);
        }
        if(primaryContacts.size()>0){
            insert primaryContacts;
        }
    }
}
```

`AddPrimaryContactTest.apxc`

```
@isTest
public class AddPrimaryContactTest {
```

```
    static testmethod void testQueueable(){
  List<Account> testAccounts = new List<Account>();
        for(Integer i=0;i<50;i++){
            testAccounts.add(new Account(Name='Account
  '+i,
                                         BillingState='CA')
  );
        }
        for(Integer j=0;j<50;j++){
            testAccounts.add(new Account(Name='Account
  '+j,
                                         BillingState='NY')
  );
        }
        insert testAccounts;
 Contact testContact = new
  Contact(FirstName='John',LastName='Doe');
 insert testContact;

        AddPrimaryContact addit = new
  addPrimaryContact(testContact,'CA');
        // startTest/stopTest block to force async
  processes to run
        Test.startTest();
        System.enqueueJob(addit);
        Test.stopTest();
        // Validate the job ran. Check if record have
  correct parentId now
        System.assertEquals(50, [select count() from
  Contact where accountId in (Select Id from Account where
  BillingState='CA')]);
 }
}
```

CHALLENGE : Schedule Jobs Using the Apex Scheduler

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

Solution :

`DailyLeadProcessor.apxc`

```
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;
        }
    }
}
```

`DailyLeadProcessorTest.apxc`

```
@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();
     }
}
```

Apex Specialist

Apex Integration Services

CHALLENGE : Apex REST Callouts

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

Solution :

`AnimalLocator.apxc`

```
public class AnimalLocator {
 public static String getAnimalNameById(Integer animalId)
  {
        String animalName;
        Http http = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndpoint('https://th-apex-http-
  callout.herokuapp.com/animals/'+animalId);
        request.setMethod('GET');
        HttpResponse response = http.send(request);
        // If the request is successful, parse the JSON
  response.
        if(response.getStatusCode() == 200) {
   Map<String, Object> r = (Map<String, Object>)
                JSON.deserializeUntyped(response.getBody()
  );
            Map<String,
  Object>animal=(Map<String,Object>)r.get('animal');
            animalName =
  string.valueOf(animal.get('name'));
        }
        return animalName;
    }
}
```

`AnimalLocatorTest.apxc`

```
@isTest
private class AnimalLocatorTest{
@isTest
static void getAnimalNameByIdTest() {
    // Set mock callout class
    Test.setMock(HttpCalloutMock.class, new
  AnimalLocatorMock());
    // This causes a fake response to be sent
    // from the class that implements HttpCalloutMock.
    String response = AnimalLocator.getAnimalNameById(1);
    // Verify that the response received contains fake
  values
    System.assertEquals('chicken', response);
}
}
```

AnimalLocatorMock.apxc

```
@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('{"animal":{"id":1,"name":"chicke
  n","eats":"chicken food","says":"cluck cluck"}}');
        response.setStatusCode(200);
        return response;
    }
}
```

CHALLENGE : Apex SOAP Callouts

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.

Solution :

`ParkLocator.apxc`

```
public class ParkLocator {
    public static List<String> country(String country){
        ParkService.ParksImplPort parkservice =
            new parkService.ParksImplPort();
        return parkservice.byCountry(country);
    }
}
```

`ParkLocatorTest.apxc`

```
@isTest
private class ParkLocatorTest {
    @isTest static void testCallout() {
        // This causes a fake response to be generated
        Test.setMock(WebServiceMock.class, new
  ParkServiceMock());
        // Call the method that invokes a callout
        String country = 'United States';
        List<String>
  result=  ParkLocator.country(country);
        List<String> parks = new List<String>();
        parks.add('Yosemite');
```

```
                parks.add('Yellowstone');
                parks.add('Another Park');
            // Verify that a fake result is returned
            System.assertEquals(parks, result);
        }
    }
```

ParkServiceMock.apxc

```
@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\
        List<String> parks = new List<string>();
                parks.add('Yosemite');
                parks.add('Yellowstone');
                parks.add('Another Park');
        ParkService.byCountryResponse response_x =
            new ParkService.byCountryResponse();
        response_x.return_x = parks;
        // end
        response.put('response_x', response_x);
    }
}
```

CHALLENGE : Apex Web Services

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

Solution :

`AccountManager.apxc`

```
@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;
    }
}
```

`AccountManagerTest.apxc`

```
@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;
    }
}
```