APEX SPECIALISTS

APEX TRIGGERS

Challenge : Get started with Apex triggers

Create an Apex trigger for Account that matches Shipping Address Postal Code with Billing Address Postal Code based on a custom field.For this challenge, you need to create a trigger that, before insert or update, checks for a checkbox, and if the checkbox field is true, sets the Shipping Postal Code (whose API name is ShippingPostalCode) to be the same as the Billing Postal Code (BillingPostalCode).The Apex trigger must be called 'AccountAddressTrigger'.The Account object will need a new custom checkbox that should have the Field Label 'Match Billing Address' and Field Name of 'Match_Billing_Address'. The resulting API Name should be 'Match_Billing_Address__c'.With 'AccountAddressTrigger' active, if an Account has a Billing Postal Code and 'Match_Billing_Address__c' is true, the record should have the Shipping Postal Code set to match on insert or update.

Solution:

```
trigger AccountAddressTrigger on Account (before insert, before update) {
   for(Account a: Trigger.New){
      if(a.Match_Billing_Address__c == true && a.BillingPostalCode!= null){
         a.ShippingPostalCode=a.BillingPostalCode;
      }
   }

}
```

Challenge : Bulk Apex Triggers Unit

Create an Apex trigger for Opportunity that adds a task to any opportunity set to 'Closed Won'.
To complete this challenge, you need to add a trigger for Opportunity. The trigger will add a task to any opportunity inserted or updated with the stage of 'Closed Won'. The task's subject must be 'Follow Up Test Task'.
The Apex trigger must be called 'ClosedOpportunityTrigger'
With 'ClosedOpportunityTrigger' active, if an opportunity is inserted or updated with a stage of 'Closed Won', it will have a task created with the subject 'Follow Up Test Task'.
To associate the task with the opportunity, fill the 'WhatId' field with the opportunity ID.
This challenge specifically tests 200 records in one operation.

APEX SPECIALISTS

solution:

```
trigger ClosedOpportunityTrigger on Opportunity (after insert, after
update) {
   List<Task> taskList = new List<Task>();

   for(Opportunity opp : [SELECT Id, StageName FROM Opportunity
WHERE StageName='Closed Won' AND Id IN : Trigger.New]){
      taskList.add(new Task(Subject='Follow Up Test Task', WhatId =
opp.Id));




   if(taskList.size()>0){
      insert tasklist;
   }

}
```

APEX SPECIALISTS

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`
- 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 test1(){
      Date d=VerifyDate.CheckDates(Date.parse('01/01/2020'),
Date.parse('01/03/2020'));
      System.assertEquals(Date.parse('01/03/2020'), d);
   }
   @isTest static void test2(){
      Date d=VerifyDate.CheckDates(Date.parse('01/01/2020'),
Date.parse('03/03/2020'));
      System.assertEquals(Date.parse('01/31/2020'), d);
   }

}
```

APEX SPECIALISTS

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`
- Place the unit tests in a separate test class
  - Name: `TestRestrictContactByName`
  - Goal: 100% test coverage
- Run your test class at least once

Solution:

RestrictedContactByName.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');
		}

	}
}
```

TestRestrictedContactByName.apxc

```
@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() );
   }

}
```

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.

- 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:

RandomContactFactory.apxc

```
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;
   }

}
```

Asynchronus Apex

challenge:Use Future Methods

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

Solution:

AccountProcessor.apxc

```
public class AccountProcessor {
   @future public static void countContacts(List<Id> accountIds){
      List<Account> accList=[Select Id,Number_Of_Contacts__c,(select Id from Contacts) from Account where Id in :accountIds];

         For(Account acc: accList){
            acc.Number_Of_Contacts__c=acc.Contacts.size();

         }
      update accList;
   }

}
```

AccountProcessorTest.apxc

```
@isTest
public class AccountProcessorTest {
   public static testmethod void testAccountProcessor(){
      Account a=new Account();
      a.Name='Test Account';
      insert a;
```

```
        Contact con=new Contact();
        con.FirstName='Binary';
        con.LastName='Programming';
        con.AccountId=a.Id;
        insert con;
        List<Id> accListId=new List<Id>();
        accListId.add(a.Id);
        Test.startTest();
        AccountProcessor.countContacts(accListId);
        Test.stopTest();
        Account acc=[Select Number_Of_Contacts__c from Account where Id =: a.Id];
        System.assertEquals(Integer.valueOf(acc.Number_Of_Contacts__c),1);
    }
}
```

challenge:Use Batch Apex

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

APEX SPECIALISTS

Solution:

LeadProcessor.apxc

```
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!');

   }
}
```

LeadProcessorTest.apxc

```
@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']);
    }
}
```

challenge:Control processes with Queuable Apex

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

Solution:

AddPrimaryContact.apxc

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

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

   public void execute(QueueableContext qc) {
      system.debug('this.c = '+this.c+' this.state = '+this.state);
      List<Account> acc_lst = new List<account>([select id, name, BillingState from account
where account.BillingState = :this.state limit 200]);
      List<contact> c_lst = new List<contact>();
      for(account a: acc_lst) {
         contact c = new contact();
         c = this.c.clone(false, false, false, false);
         c.AccountId = a.Id;
         c_lst.add(c);
      }
      insert c_lst;
   }

}
```

TestAddPrimaryContact.apxc

```
@IsTest
public class AddPrimaryContactTest {

    @IsTest
    public static void testing() {
        List<account> acc_lst = new List<account>();
        for (Integer i=0; i<50;i++) {
            account a = new account(name=string.valueOf(i),billingstate='NY');
            system.debug('account a = '+a);
            acc_lst.add(a);
        }
        for (Integer i=0; i<50;i++) {
            account a = new
account(name=string.valueOf(50+i),billingstate='CA');
            system.debug('account a = '+a);
            acc_lst.add(a);
        }
        insert acc_lst;
        Test.startTest();
        contact c = new contact(lastname='alex');
        AddPrimaryContact apc = new AddPrimaryContact(c,'CA');
        system.debug('apc = '+apc);
        System.enqueueJob(apc);
        Test.stopTest();
        List<contact> c_lst = new List<contact>([select id from contact]);
        Integer size = c_lst.size();
        system.assertEquals(50, size);
    }

}
```

APEX SPECIALISTS

challenge:Schedule jobs using the Apex Scheduler

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

Solution:

DailyLeadProcessor.apxt

```
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;
      }
   }
}
```

APEX SPECIALISTS

DailyLeadProcessorTest.apxt

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

challenge: Apex REST callouts

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

solution:

AnimalLocator.apxt:

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

APEX SPECIALISTS

AnimalLocatorTest.apxt:

```
@istest
private class AnimalLocatorTest {
@isTest static void getAnimalNameById() {
   // 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);
}
}
```

challenge:Apex SOAP callouts

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.

Solution:

ParkLocator.apxt

```
public class ParkLocator {
   public static List<string> country(string country){
      ParkService.ParksImplport parkservice=
         new parkService.parksImplPort();
      return parkservice.Country(country);
}
}
```

ParkLocatorTest.apxt

```
@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);
   }
}
```

APEX SPECIALISTS

challenge:Apex Web Service

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

Solution:

AccountManager.apxt

```
@RestResource(urlMapping='/Accounts/*/contacts')
global with sharing class AccountManager {
   @HttpGet
   global static Account getAccount() {
      RestRequest request = RestContext.request;
      // grab the caseId from the end of the URL
      String AccountId = request.requestURI.substringBetween('Accounts/','/contacts');
      Account result =  [SELECT Id,Name, (select Id,Name from Contacts) from Account where
Id=:accountId];
      return result;
   }
}
```

APEX SPECIALISTS

AccountManagerTest.apxt

```
@IsTest
private class AccountManagerTest {
    @isTest static void testGetContactsByAccountId() {
        Id recordId = createTestRecord();
        // Set up a test request
        RestRequest request = new RestRequest();
        request.requestUri =
            'https://yourInstance.my.salesforce.com/services/apexrest/Accounts/'
            + recordId+'/contacts';
        request.httpMethod = 'GET';
        RestContext.request = request;
        // Call the method to test
        Account thisAccount = AccountManager.getAccount();
        // Verify results
        System.assert(thisAccount != null);
        System.assertEquals('Test record', thisAccount.Name);
    }

    // Helper method
    static Id createTestRecord() {
        // Create test record
        Account accountTest = new Account(
            Name='Test record');
             insert accountTest;
        Contact contactTest=new Contact(FirstName='John',
                                LastName='Doe',
                                AccountId=accountTest.Id);
        insert contactTest;
        return accountTest.Id;
    }
}
```