APEX TRIGGERS

GET STARTED WITH APEX TRIGGERS

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.

Trigger Class: AccountAddressTrigger.apxt

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

}
```

APEX TRIGGERS
BULK APEX TRIGGERS

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.

Trigger Class: ClosedOppourtunityTrigger.apxt

```
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 TESTING

GET STARTED WITH APEX UNIT TESTS

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.

Apex Class: 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;
  }

}
```

APEX TESTING

TEST APEX TRIGGERS UNIT

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.

Apex Class: RestrictContactByName.apxc

```
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');
        }

    }



}
```

APEX TESTING

CREATE TEST DATA FOR APEX TESTS

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.

Apex Class:RandomContactFactory.apxc

```
public class RandomContactFactory {
    public static List<Contact> generateRandomContacts(Integer
numContactsToGenerate, String FName) {
        List<Contact> contactList = new List<Contact>();
        
        for(Integer i=0;i<numContactsToGenerate;i++) {
            Contact c = new Contact(FirstName=FName + ' ' + i, LastName = 'Contact '+i);
            contactList.add(c);
            System.debug(c);
        }


        //insert contactList;



        System.debug(contactList.size());
        return contactList;
    }

}
```

ASYNCHRONOUS APEX

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 to hold the total number of Contacts for the Account

Apex Class: AccountProcessor.apxc

```
public class AccountProcessor {
    @future
    public static void countContacts(List<Id> accountIds){
        List<Account> accounts = [Select Id, Name from Account Where Id IN :
accountIds];
        List<Account> updatedAccounts = new List<Account>();
        for(Account account : accounts){
           account.Number_of_Contacts__c = [Select count() from Contact Where
AccountId =: account.Id];
            System.debug('No Of Contacts = ' + account.Number_of_Contacts__c);
            updatedAccounts.add(account);
        }
        update updatedAccounts;
    }

}
```

Apex Test Class: AccountProcessorTest.apxc

```
@isTest
public class AccountProcessorTest {
    @isTest
    public static void testNoOfContacts(){
        Account a = new Account();
        a.Name = 'Test Account';
        Insert a;
        
        Contact c = new Contact();
        c.FirstName = 'Bob';
        c.LastName =  'Willie';
        c.AccountId = a.Id;
        
        Contact c2 = new Contact();
        c2.FirstName = 'Tom';
        c2.LastName = 'Cruise';
        c2.AccountId = a.Id;
        
        List<Id> acctIds = new List<Id>();
        acctIds.add(a.Id);
        
        Test.startTest();
        AccountProcessor.countContacts(acctIds);
        Test.stopTest();
    }

}
```

ASYNCHRONOUS APEX

USE BATCH APEX UNIT

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.

Apex Class: LeadProcessor.apxc

```
public class LeadProcessor implements Database.Batchable<sObject> {

    public Database.QueryLocator start(Database.BatchableContext bc) {


        // collect the batches of records or objects to be passed to execute


          return Database.getQueryLocator([Select LeadSource From Lead ]);
    }
    public void execute(Database.BatchableContext bc, List<Lead> leads){


         // process each batch of records


            for (Lead Lead : leads) {
                lead.LeadSource = 'Dreamforce';
            }
        update leads;
    }
    public void finish(Database.BatchableContext bc){
      }
}
```

Apex Test Class: LeadProcessorTest.apxc

```
@isTest
public class LeadProcessorTest {

        @testSetup
    static void setup() {
        List<Lead> leads = new List<Lead>();
        for(Integer counter=0 ;counter <200;counter++){
            Lead lead = new Lead();
            lead.FirstName ='FirstName';
            lead.LastName ='LastName'+counter;
            lead.Company ='demo'+counter;
            leads.add(lead);
        }
        insert leads;
    }

    @isTest static void test() {
        Test.startTest();
        LeadProcessor leadProcessor = new LeadProcessor();
        Id batchId = Database.executeBatch(leadProcessor);
        Test.stopTest();
    }

}
```

ASYNCHRONOUS APEX

COUNT PROCESSES WITH QUEUEABLE 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.

Apex Class: AddPrimaryContact.apxc

```
public class AddPrimaryContact implements Queueable
{
    private Contact c;
    private String state;
    public  AddPrimaryContact(Contact c, String state)
    {
        this.c = c;
        this.state = state;
    }
    public void execute(QueueableContext context)
    {
         List<Account> ListAccount = [SELECT ID, Name ,(Select id,FirstName,LastName
from contacts ) FROM ACCOUNT WHERE BillingState = :state LIMIT 200];
         List<Contact> lstContact = new List<Contact>();
         for (Account acc:ListAccount)
         {
                 Contact cont = c.clone(false,false,false,false);
                 cont.AccountId =  acc.id;
                 lstContact.add( cont );
         }

         if(lstContact.size() >0 )
         {
             insert lstContact;
        }

    }

}
```

Apex Test Class: AddPrimaryContactTest.apxc

```
@isTest
public class AddPrimaryContactTest
{
     @isTest static void TestList()
     {
         List<Account> Teste = new List <Account>();
         for(Integer i=0;i<50;i++)
         {
             Teste.add(new Account(BillingState = 'CA', name = 'Test'+i));
         }
         for(Integer j=0;j<50;j++)
         {
             Teste.add(new Account(BillingState = 'NY', name = 'Test'+j));
         }
         insert Teste;

         Contact co = new Contact();
         co.FirstName='demo';
         co.LastName ='demo';
         insert co;
         String state = 'CA';

          AddPrimaryContact apc = new AddPrimaryContact(co, state);
          Test.startTest();
            System.enqueueJob(apc);
          Test.stopTest();
      }
 }
```

ASYNCHRONOUS APEX

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.)

Apex Class: DailyLeadProcessor.apxc

```
public class DailyLeadProcessor implements Schedulable  {
    Public void execute(SchedulableContext SC){
       List<Lead> LeadObj=[SELECT Id from Lead where LeadSource=null limit 200];
        for(Lead l:LeadObj){
            l.LeadSource='Dreamforce';
            update l;
        }
    }
}
```

Apex Test Class: DailyLeadProcessorTest.apxc

```
@isTest
private class DailyLeadProcessorTest {
  static testMethod void testDailyLeadProcessor() {
    String CRON_EXP = '0 0 1 * * ?';
    List<Lead> lList = new List<Lead>();
      for (Integer i = 0; i < 200; i++) {
      lList.add(new Lead(LastName='Dreamforce'+i, Company='Test1 Inc.',
Status='Open - Not Contacted'));
    }
    insert lList;

    Test.startTest();
    String jobId = System.schedule('DailyLeadProcessor', CRON_EXP, new
DailyLeadProcessor());
  }
}
```

APEX INTEGRATION SERVICES

APEX **REST** CALLOUTS UNIT

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.

Apex Class: AnimalLocator.apxc

```
public class AnimalLocator{
    public static String getAnimalNameById(Integer x){
        Http http = new Http();
        HttpRequest req = new HttpRequest();
        req.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals/' + x);
        req.setMethod('GET');
        Map<String, Object> animal= new Map<String, Object>();
        HttpResponse res = http.send(req);
            if (res.getStatusCode() == 200) {
        Map<String, Object> results = (Map<String,
Object>)JSON.deserializeUntyped(res.getBody());
      animal = (Map<String, Object>) results.get('animal');
        }
return (String)animal.get('name');
    }
}
```

Apex Test Class: AnimalLocatorTest.apxc

```
@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 );
    }
}
```

APEX INTEGRATION SERVICES

APEX **SOAP** CALLOUTS UNIT

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.

Apex Class: ParkLocator.apxc

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

Apex Test Class:ParkLoactorTest.apxc

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

APEX INTEGRATION SERVICES

APEX WEB SERVICES UNIT

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.

Apex Class: AccountManager.apxc

```
@RestResource(urlMapping='/Accounts/*/contacts')
global class AccountManager {
    @HttpGet
    global static Account getAccount() {
        RestRequest req = RestContext.request;
        String accId = req.requestURI.substringBetween('Accounts/', '/contacts');
        Account acc = [SELECT Id, Name, (SELECT Id, Name FROM Contacts)
                       FROM Account WHERE Id = :accId];
        return acc;
    }
}
```

Apex Test Class: AccountManagerTest.apxc

```
@isTest
private class AccountManagerTest {

    private static testMethod void getAccountTest1() {
        Id recordId = createTestRecord();
        
        RestRequest request = new RestRequest();
        request.requestUri = 'https://na1.salesforce.com/services/apexrest/Accounts/'+
recordId +'/contacts' ;
        request.httpMethod = 'GET';
        RestContext.request = request;
        
        Account thisAccount = AccountManager.getAccount();
        
        System.assert(thisAccount != null);
        System.assertEquals('Test record', thisAccount.Name);

    }

    
        static Id createTestRecord() {
        
        Account TestAcc = new Account(
          Name='Test record');
        insert TestAcc;
        Contact TestCon= new Contact(
        LastName='Test',
        AccountId = TestAcc.id);
        return TestAcc.Id;
    }
}
```