Apex Triggers

1. Get Started With Apex triggers

TASK -

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, before update) {
        for(Account a : Trigger.New)
   {
      if(a.Match_Billing_Address__c == True)
      {
         a.ShippingPostalCode = a.BillingPostalCode;
      }
   }
}
```

2. Bulk Apex Triggers

TASK

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 (after insert, after update) {
        List<Task> tl = new List<Task>();
   for(Opportunity opp: Trigger.New)
   {
      if(opp.StageName == 'Closed Won')
      {
         tl.add(new Task(Subject = 'Follow Up Test Task', WhatId = opp.Id));
      }
   }
   if(tl.size() > 0)
   {
      insert tl;
   }
}
```

Apex testing

Get Started With APEX unit tests

TASK -

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 -

```
@isTest
public class TestVerifyDate {
  @isTest
   static void checkdates_1()
   {
      Date ans = VerifyDate.CheckDates(date.parse('01/02/2022'), date.parse('01/05/2022'));
      System.assertEquals(date.parse('01/05/2022'), ans);
   }
   @isTest
   static void checkdates_2()
   {
      Date ans = VerifyDate.CheckDates(date.parse('01/02/2022'), date.parse('02/05/2022'));
      System.assertEquals(date.parse('01/31/2022'), ans);
   }
   @isTest
   static void DateWithin30_1()
   {
      Date ans = VerifyDate.CheckDates(date.parse('01/02/2022'), date.parse('12/01/2022'));
      System.assertEquals(date.parse('01/31/2022'), ans);
   }
   @isTest
   static void DateWithin30_2()
   {
      Date ans = VerifyDate.CheckDates(date.parse('01/02/2022'), date.parse('03/01/2022'));
      System.assertEquals(date.parse('01/31/2022'), ans);
   }
}
```

Test Apex Triggers
Task -

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 -

```
@isTest
public class TestRestrictContactByName {

   @isTest
   static void invalid_case()
   {
      Contact ct = new Contact();
      ct.LastName = 'INVALIDNAME';

      Test.startTest();
      Database.SaveResult result = Database.insert(ct, false);
      Test.stopTest();


  System.assert(!result.isSuccess());
  System.assert(result.getErrors().size() > 0);
  System.assertEquals('The Last Name "INVALIDNAME" is not allowed for DML',
result.getErrors()[0].getMessage());
   }
}
```

Create test Data for Apex tests

TASK -

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 -

```
public class RandomContactFactory {
        public static List<Contact> generateRandomContacts(Integer num_contacts,
String l_name)
   {
      List<Contact> cont = new List<Contact>();
      for(integer i = 0;i<num_contacts;i++)
      {
         Contact ct = new Contact();
         ct.FirstName = ''+i;
         ct.LastName = l_name;
         cont.add(ct);
      }
      return cont;
   }
}
```

LIghtning Web Components
TASK -
Create an app page for the bike card component

Deploy your files to your Trailhead Playground or Developer Edition org and then use Lightning App Builder to create an app page.

Prework: If you haven't already completed the activities in the What You Need section of this unit, do that now, or this challenge won't pass. Make sure that both Dev Hub and My Domain are enabled in your org and that the org is authorized with Visual Studio Code.

- Create an SFDX project in Visual Studio Code:
    - Template: **Standard**
    - Project name: bikeCard
- Add a Lightning Web Component to the project:
    - Folder: **lwc**
    - Component name: bikeCard
- Copy the content for the component files from this unit into your files in Visual Studio Code:
    - bikeCard.html
    - bikeCard.js
    - bikeCard.js-meta.xml
- Deploy the bikeCard component files to your org
- Create a Lightning app page:
    - Label: Bike Card
    - Developer Name: Bike_Card
    - Add your bikeCard component to the page
    - Activate the page for all users

Code -
bikeCard.js

```
import { LightningElement } from 'lwc';
export default class BikeCard extends LightningElement {
  name = 'Electra X4';
  description = 'A sweet bike built for comfort.';
  category = 'Mountain';
  material = 'Steel';
  price = '$2,700';
  pictureUrl = 'https://s3-us-west-1.amazonaws.com/sfdc-demo/ebikes/electrax4.jpg';
}
```

bikeCard.html

```
<template>
    <div>
        <div>Name: {name}</div>
        <div>Description: {description}</div>
        <lightning-badge label={material}></lightning-badge>
        <lightning-badge label={category}></lightning-badge>
        <div>Price: {price}</div>
        <div><img src={pictureUrl} /></div>
    </div>
</template>
```

bikeCard.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <!-- The apiVersion may need to be increased for the current release -->
    <apiVersion>54.0</apiVersion>
    <isExposed>true</isExposed>
    <masterLabel>Product Card</masterLabel>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
```

```
</LightningComponentBundle>
```

Add Styles and Data to a Lightning Web Component

TASK -
Create a Lightning app page that uses the wire service to display the current user's name.

Prework: You need files created in the previous unit to complete this challenge. If you haven't already completed the activities in the previous unit, do that now.

- Create a Lightning app page:
  - Label: Your Bike Selection
  - Developer Name: Your_Bike_Selection
- Add the current user's name to the app container:
  - Edit selector.js
  - Edit selector.html

CODE -
Selector.js

```
import { LightningElement, wire } from 'lwc';
import { getRecord, getFieldValue } from 'lightning/uiRecordApi';
import Id from '@salesforce/user/Id';
import NAME_FIELD from '@salesforce/schema/User.Name';
const fields = [NAME_FIELD];
export default class Selector extends LightningElement {
    selectedProductId;
    handleProductSelected(evt) {
        this.selectedProductId = evt.detail;
    }
    userId = Id;
    @wire(getRecord, { recordId: '$userId', fields })
    user;
    get name() {
        return getFieldValue(this.user.data, NAME_FIELD);
    }
}
```

Selector.html

```
<template>
    <div class="wrapper">
        <header class="header">Available Bikes for {name}</header>
        <section class="content">
            <div class="columns">
                <main class="main">
                    <c-list onproductselected={handleProductSelected}></c-list>
                </main>
                <aside class="sidebar-second">
                    <c-detail product-id={selectedProductId}></c-detail>
                </aside>
            </div>
        </section>
    </div>
</template>
```

Asynchronous Apex-
Use Future methods
Task -

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(List<Id> accountIds)
   {
      List<Account> accToUp = new List<Account>();
      List<Account> acc = [Select Id, Name, (Select Id from Contacts) from Account where Id
in :accountIds];
      For(Account a:acc)
      {
         List<Contact> contList =  a.Contacts;
         a.Number_Of_Contacts__c = contList.size();
         accToUp.add(a);
      }
      update accToUp;
   }
}
```

AccountProcessorTest

```
@isTest
private class AccountProcessorTest {
        @isTest
   private static void testCountCont()
   {
      Account newAccount = new Account(Name = 'Testing Account');
      insert newAccount;
      Contact newContact_one = new Contact(FirstName ='TestcontactsCount',
LastName='One', AccountId = newAccount.Id);
      insert newContact_one;

      Contact newContact_two = new Contact(FirstName ='TestcontactsCount',
LastName='Two', AccountId = newAccount.Id);
      insert newContact_two;

      List<Id> accountIds = new List<Id>();
      accountIds.add(newAccount.Id);

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

Use Batch Apex
TASK -

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 -
LeadProcessor

```
public class LeadProcessor implements Database.Batchable<sObject>{
        public integer c=0;
   public Database.QueryLocator start(Database.BatchableContext bc)
   {
        String query = 'SELECT Id, LeadSource from Lead';
      return Database.getQueryLocator(query);
   }
   public void execute (Database.BatchableContext bc, List<Lead> l_list)

   {
      List<lead> l_list_new = new List<lead>();
      for(lead L : l_list){
         L.leadsource = 'Dreamforce';
         l_list_new.add(L);
         c+=1;
      }
      update l_list;
   }
   public void finish(Database.BatchableContext bc)
   {
      System.debug('count = '+c);
   }
}
```

LeadProcessorTest

```
@isTest public class LeadProcessorTest {
```

```
        @isTest
    public static void test_lead(){
        List<lead> l_list = new List<lead>();
        for(integer i = 0;i<200;i++)
        {
            Lead L = new lead();
            L.LastName = 'John'+i;
            L.Company = 'Company';
            L.Status = 'Random Status';
            l_list.add(L);
        }
        insert l_list;
        Test.startTest();
        LeadProcessor l_p = new LeadProcessor();
        Id batchId = Database.executeBatch(l_p);
        Test.stopTest();
    }
}
```

ControlProcessing with queueable apex

TASK -

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-

Class - AddPrimaryContact

```
public class AddPrimaryContact implements Queueable{
        private Contact ct;
   private string state_val;

   public AddPrimaryContact(Contact ct, string state_val)
   {
      this.ct = ct;
      this.state_val = state_val;
   }
   public void execute(QueueableContext context){
      List<Account> acc = [Select Id, Name,(Select FirstName, LastName, Id from contacts) from
Account where BillingState = :state_val Limit 200];
      List<Contact> main_contacts = new List<Contact>();

      for(Account acct:acc){
         Contact c= ct.clone();
         c.AccountId = acct.Id;
         main_contacts.add(c);
      }
      if(main_contacts.size() > 0)
         {
            insert main_contacts;
         }
   }
}
```

Class - AddPrimaryContactTest

```
@isTest public class AddPrimaryContactTest {
   static testmethod  void testQueueable(){

      List<Account> acc_for_test = new List<Account>();
      for(integer i= 0;i<50;i++)
      {
         acc_for_test.add(new Account(Name = 'john'+i, BillingState = 'NY'));
      }
      for(integer i= 0;i<50;i++)
      {
         acc_for_test.add(new Account(Name = 'Colossal'+i, BillingState = 'CA'));
      }
      insert acc_for_test;

      Contact test_c = new Contact(FirstName = 'James', LastName='Duke');
      insert test_c;

      AddPrimaryContact apc = new AddPrimaryContact(test_c, 'CA');
      Test.startTest();
      System.enqueueJob(apc);
      Test.stopTest();
      System.assertEquals(50, [Select count() from Contact where accountId in (Select Id from
Account where BillingState='CA')]);
   }
}
```

Schedule Jobs Using Apex Scheduler

TASK -

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 -

Class - DailyLeadProcessor

```
public class DailyleadProcessor implements Schedulable {
   public void execute(SchedulableContext ctx) {
      List<lead> l_list = new List<lead>();
      List<Lead> leads = [Select id From Lead Where LeadSource=Null Limit 200];

      for(Lead l : leads){
         l.LeadSource = 'Dreamforce';
         l_list.add(l);
      }
      update l_list;
   }
}
```

Class - DailyLeadProcessorTest

```
@isTest public class DailyLeadProcessorTest {
         public static String CRON_EXP = '0 0 0 15 10 ? 2022';
   static testmethod void test_DailyLeadProcessor()
   {
      List <Lead> leads = new List<lead>();
      for(integer i = 0;i<200;i++)
      {
         Lead l = new Lead(FirstName = 'name'+i, LastName = 'bidon', Company = 'The Company');
         leads.add(l);
      }
      insert leads;
      Test.startTest();
```

```
        String JobId = System.schedule('ScheduledApexTest', CRON_EXP, new
DailyLeadProcessor());
        Test.stopTest();
        List<Lead> check = new List<Lead>();
        check = [Select Id from Lead Where LeadSource='Dreamfroce' AND Company = 'The
Company'];
        System.assertEquals(200, check.size(), 'Not created');
    }
}
```

Apex REST Callouts
TASK -

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 -
AnimalLocator

```
public class AnimalLocator {
    public static String getAnimalNameById(Integer value) {
```

```
        String Name;
        Http http = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals'+value);
        request.setMethod('GET');
        HttpResponse response = http.send(request);
        if(response.getStatusCode() == 200) {
            Map<String, Object> results = (Map<String, Object>)
JSON.deserializeUntyped(response.getBody());
            Map<String, Object> entity = (Map<String, Object>)results.get('animal');

            Name = string.valueOf(entity.get('name'));
        }
        return Name;
    }
}
```

AnimalLocatorMock

```
@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":2,"name":"bear","eats":"berries, campers, adam
seligman","says":"yum yum"}}');
        response.setStatusCode(200);
        return response;
    }
}
```

AnimalLocatorTest

```
@isTest
private class AnimalLocatorTest{
    @isTest static void getAnimalNameById_test()
    {
        Test.setMock(HttpCalloutMock.class, new AnimalLocatorMock());
        String res = AnimalLocator.getAnimalNameById(1);
        System.assertEquals('bear', res);
    }
}
```

Apex SOAP Callouts:

Task -

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 -

ParkService

```
//Generated by wsdl2apex

public class ParkService {
    public class byCountryResponse {
        public String[] return_x;
        private String[] return_x_type_info = new
String[]{'return','http://parks.services/',null,'0','-1','false'};
        private String[] apex_schema_type_info = new
String[]{'http://parks.services/','false','false'};
        private String[] field_order_type_info = new String[]{'return_x'};
    }
    public class byCountry {
        public String arg0;
        private String[] arg0_type_info = new
String[]{'arg0','http://parks.services/',null,'0','1','false'};
        private String[] apex_schema_type_info = new
String[]{'http://parks.services/','false','false'};
```

```
        private String[] field_order_type_info = new String[]{'arg0'};
    }
    public class ParksImplPort {
        public String endpoint_x = 'https://th-apex-soap-
service.herokuapp.com/service/parks';
        public Map<String,String> inputHttpHeaders_x;
        public Map<String,String> outputHttpHeaders_x;
        public String clientCertName_x;
        public String clientCert_x;
        public String clientCertPasswd_x;
        public Integer timeout_x;
        private String[] ns_map_type_info = new String[]{'http://parks.services/',
'ParkService'};
        public String[] byCountry(String arg0) {
            ParkService.byCountry request_x = new ParkService.byCountry();
            request_x.arg0 = arg0;
            ParkService.byCountryResponse response_x;
            Map<String, ParkService.byCountryResponse> response_map_x = new
Map<String, ParkService.byCountryResponse>();
            response_map_x.put('response_x', response_x);
            WebServiceCallout.invoke(
              this,
              request_x,
              response_map_x,
              new String[]{endpoint_x,
              '',
              'http://parks.services/',
              'byCountry',
              'http://parks.services/',
              'byCountryResponse',
              'ParkService.byCountryResponse'}
            );
            response_x = response_map_x.get('response_x');
```

```
        return response_x.return_x;
      }
   }
}
```

ParkLocator

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

ParkServiceMock

```
@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) {
              List<String> p = new List<String>();
       p.add('Anamudi Shola National Park');
              p.add('Anshi National Park');
           p.add('Bandipur National Park');
      ParkService.byCountryResponse response_x =
         new ParkService.byCountryResponse();
      response_x.return_x = p;
      response.put('response_x', response_x);
```

```
  }
}
```

Park Locator Test

```
@isTest
private class ParkLocatorTest {
   @isTest static void testCallout() {
      Test.setMock(WebServiceMock.class, new ParkServiceMock());
      List<String> p = new List<String>();
        p.add('Anamudi Shola National Park');
              p.add('Anshi National Park');
      p.add('Bandipur National Park');
        List<String> res = ParkLocator.country('India');
      System.assertEquals(p, res);
   }
}
```

Apex Web Services

TASK -

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 -

AccountManager

```
@RestResource(urlMapping='/Accounts/*/contacts')
global with sharing class AccountManager {
    @HttpGet
    global static Account getAccount() {
        RestRequest request = RestContext.request;
        // grab the AccountId from the mid 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 Limit 1];
        return result;
    }
}
```

AccountManagerTest

```
@isTest
private class AccountManagerTest {
    @isTest static void getContactsByAccountIdTest(){
        Id AccountId = createTestRecord();
        RestRequest request = new RestRequest();
        request.requestURi =
'https://yourInstance.my.salesforce.com/services/apexrest/Accounts/'+AccountId
+'/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 c = new Contact(FirstName='Shima', LastName='Doe', AccountId =
testacc.Id);
        insert c;
        return testacc.Id;
    }
}
```