Apex Integration Services

(1) Apex Integration Overview.

Be sure Remote Site Settings have been added

Add two Remote Sites to the Remote Site Settings to allow callouts to external sites.

Prework: If you already added the Remote Sites as directed in the unit then you are ready to verify.

- Add a Remote Site
  - Name: `animals_http`
  - Remote site URL: `https://th-apex-http-callout.herokuapp.com`
  - Description: `Trailhead animal service: HTTP`
- Add a Remote Site
  - Name: `animals_soap`
  - Remote site URL: `https://th-apex-soap-service.herokuapp.com`
  - Description: `Trailhead animal service: SOAP`

Steps taken to complete this challenge:

(1) After launching a Salesforce Playground or an org, click on "Gear" icon on top right-side corner and then click Setup.
(2) After getting inside Setup Menu, search for "Remote Site Settings" in Quick Find Box.
(3) Click on Remote Site Settings and then click on New Remote Site button and then filled the given details, i.e., Remote Site Name, Site URL and Description in that menu.
(4) Repeat Step-3 for filling all the given details for second given site and then save both the Remote Sites and then click on Check Challenge, the task will be then successfully completed and submitted.

(2) 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

Steps taken to complete this challenge:

(1) After launching a Salesforce Playground or an org, click on "Gear" icon on top right-side corner and then click Setup.

(2) After getting inside Setup Menu, search for "Remote Site Settings" in Quick Find Box.

(3) Click on Remote Site Settings and then click on New Remote Site button and then filled the given details, i.e., Remote Site Name and Remote Site URL as "AnimalLocator" and "https://th-apex-http-callout.herokuapp.com/animals/id" respectively over there.

(4) Then again, click on gear icon and click "Developer Console" and create three new Apex classes named as "AnimalLocator", "AnimalLocatorTest" and "AnimalLocatorMock".

(5) Code needs to be written in AnimalLocator.apxc file:

```
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');
      }
    }
```

(6) Code needs to be written in AnimalLocatorTest.apxc file:

```
    @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 );
      }
    }
```

(7) Code needs to be written in AnimalLocatorMock.apxc file:

```
    @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('{"animals": ["majestic badger", "fluffy bunny", "scary
  bear", "chicken", "mighty moose"]}');
        response.setStatusCode(200);
        return response;
      }
    }
```

(8) Save all the Apex code files and click on Test and then, click on “Run All” and then click on “Check Challenge”, the task will be then successfully completed and submitted.

(3) 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.

Steps taken to complete this challenge:

(1) After launching a Salesforce Playground or an org, click on "Gear" icon on top right-side corner and then click Setup.

(2) After getting inside Setup Menu, search for "Remote Site Settings" in Quick Find Box.

(3) Click on Remote Site Settings and then click on New Remote Site button and then filled the given details, i.e., Remote Site Name and Remote Site URL as "ParkService" and "https://th-apex-soap-service.herokuapp.com" respectively over there.

(4) Then search for "Apex Classes" in Quick Find Box and click on that and then click on "Generate from WSDL" button and then click Choose File and chose the downloaded WSDL file, then click on "Parse WSDL" button and then type "ParkService" in the Apex Class Name box and click on "Generate Apex Code" and click on Done.

(5) Then again, click on gear icon and click "Developer Console" and create three new Apex classes named as "ParkLocator", "ParkLocatorTest" and "ParkServiceMock".

(6) Code needs to be written in ParkLocator.apxc file:

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

(7) Code needs to be written in ParkLocatorTest.apxc file:

```
@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);
   }
}
```

(8) Code needs to be written in ParkServiceMock.apxc file:

```
@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
      ParkService.byCountryResponse response_x = new
ParkService.byCountryResponse();
      response_x.return_x = new List<String>{'Yellowstone', 'Mackinac National
Park', 'Yosemite'};
      // end
      response.put('response_x', response_x);
  }
}
```

(9) Save all the Apex code files and click on Test and then, click on “Run All” and then click on “Check Challenge”, the task will be then successfully completed and submitted.

(4) Apex Web Services.

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

Steps taken to complete this challenge:

(1) After launching a Salesforce Playground or an org, click on “Gear” icon on top right-side corner and then click on “Developer Console”.

(2) After coming onto Developer console, create two new Apex Classes named as “AccountManager” and “AccountManagerTest” respectively.

(3) Code needs to be written in AccountManager.apxc file:

```
@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;
    }
}
```

(4) Code needs to be written in AccountManagerTest.apxc file:

```
@isTest
private class AccountManagerTest {
    private static testMethod void getAccountTest1() {
        Id recordId = createTestRecord();
        // Set up a test request
        RestRequest request = new RestRequest();
        request.requestUri =
'https://na1.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 TestAcc = new Account(
          Name='Test record');
        insert TestAcc;
        Contact TestCon= new Contact(
        LastName='Test',
        AccountId = TestAcc.id);
        return TestAcc.Id;
    }
}
```

(5) Save both of the Apex code files and click on Test and then, click on “Run All” and then click on “Check Challenge”, the task will be then successfully completed and submitted.