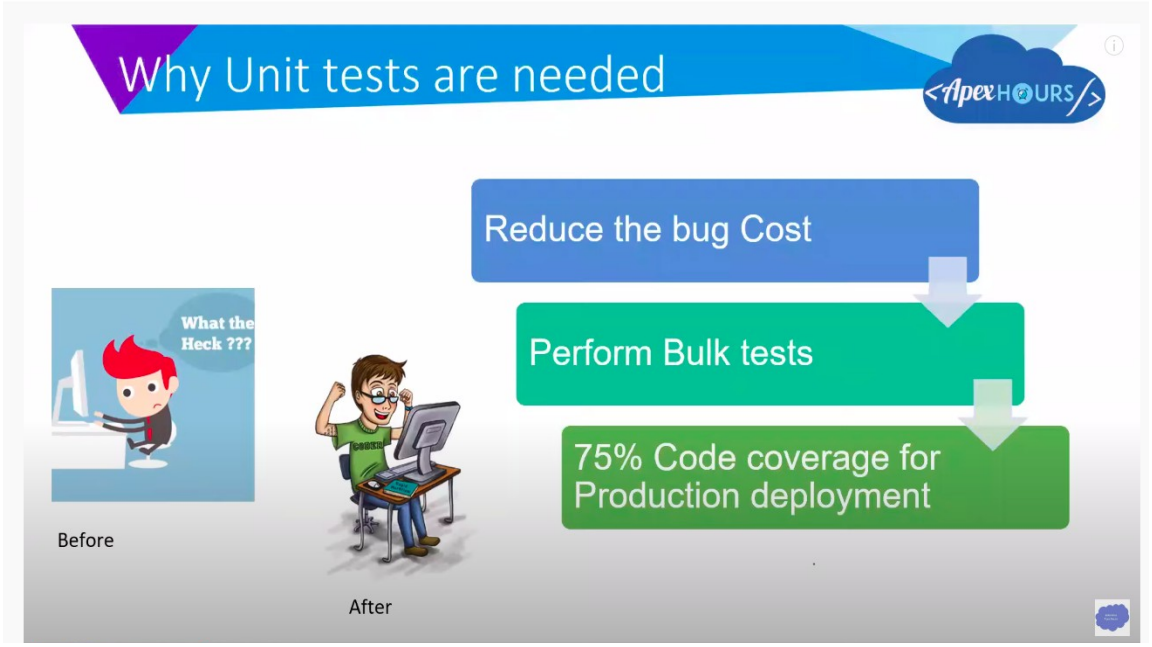
Test Classes:

The Apex testing framework enables you to write and execute tests for your Apex classes and triggers on the Lightning Platform.

Apex unit tests ensure high quality for your Apex code and let you meet requirements for deploying Apex.

Testing is the key to successful long-term development and is a critical component of the development process.

Why Unit Test are required:



- For moving the code from Sandbox to Production, atleast 75% code coverage is required, else, Production doesn't allow to to migrate the code.

- Upon deploying code from one Sandbox to another Sandbox, there is no need to maintain code coverage.

- Test classes code is not counted under 60,000,00 character limit of Apex code.

- In order to write test class, "@isTest" annotation is used, which indicates that, the class is test class.

- Access specifier of Test class is either "Private" or "Public" only.

- We can define one or more methods in a Test class to test application functionality.

- "@isTest" annotation must be used to specify that, the method is Test method.

- All methods declared inside a Test classes must be "static".

- Return type of all methods of Test classes must be "void".

- Whatever records are created in Test class, those records are stored in object till Test class execution is done. Once Test class exection is done, all records which are created in Test class, are removed automatically.

- System.debug are not counted as part of Apex code coverage.

- Every trigger you are trying to deploy should have at least 1% coverage, but overall coverage of your production org after getting your code deployed should be 75%, otherwise Salesforce wont let you deploy your code.

- Class can be deployed on 0% coverage as well, but overall coverage of your production org after getting your code deployed should be 75%, otherwise Salesforce wont let you deploy your code.

```
@isTest
public class MyTestClass {
@isTest
public static void myTest() { // code_block }

 }
```

Use Case1 :

Create an Apex Class to perform the Mathematical Operations and Prepare the Test Class to test the Business Logic.

Class Code:

```
public class MathOperationsUtility
{
    Public static void Addition(integer x, integer y, integer z)
   {
    system.debug('Addition Result : '+ (x + y + z));
   }

   Public static void Multiply(integer x, integer y)
   {
    system.debug('Multiply Result is....: '+ (x * y));
   }

   Public static void Division(integer x, integer y)
   {
    system.debug('Division Result is....: '+ (x / y));
   }

   Public static void Modulus(integer x, integer y)
```

```
    {
        system.debug('Reminder Value is....: '+ Math.mod(x, y));
    }
}
```

Test Class Code:

```
        @isTest
        Private class MathOperationsUtilityTest
        {
                @isTest static void TestMathOperations()
                {
                        // Invoke the Business Logic Methods..
                        MathOperationsUtility.Addition(100,4500,500);

                        MathOperationsUtility.Multiply(20000, 400);

                        MathOperationsUtility.Division(4500, 20);

                        MathOperationsUtility.Modulus(400,6);


                }
        }
```

UseCase 2:

```
trigger AccountTriggerNew on Account (before insert) {
    if(trigger.isBefore){
        if(Trigger.isInsert){
            AccountTriggerHandler.handleBeforeInsert(Trigger.new);
        }
    }
}
```

```
public class AccountTriggerHandler {
    public static void handleBeforeInsert(List<Account>
accountlist){
        for(Account acc:accountlist){
            if(acc.description== Null || acc.description == ''){
                acc.description='Description added from trigger';
            }

        }
    }

}
```

Test Class:

```
@isTest
public class AccountTriggerTest  {
@isTest
    public static void testInsertAccount(){
        Account acc=new Account(Name='New Test Record');
        Test.startTest();
        INSERT acc;
        Test.stopTest();
        Account InsertedAcc=[select Description from Account
where id=:acc.id];
        system.assertEquals('Description added from trigger',
InsertedAcc.Description);
    }
}
```

UseCase 3:

If an account record has related opportunities, the Account Deletion trigger prevents the record’s deletion.

```
trigger AccountDeletion on Account (before delete) {
 // Prevent the deletion of accounts if they have related
opportunities.
for (Account a : [SELECT Id FROM Account WHERE Id IN
(SELECT AccountId FROM Opportunity) AND Id
IN :Trigger.old]) {
 Trigger.oldMap.get(a.Id).addError( 'Cannot delete account with
related opportunities.');
 }
 }
```

Test Class:

```
@isTest private class TestAccountDeletion {
@isTest

static void TestDeleteAccountWithOneOpportunity() {
// Test data setup // Create an account with an opportunity, and
then try to delete it
Account acct = new Account(Name='Test Account');
insert acct;
Opportunity opp = new Opportunity(Name=acct.Name + '
Opportunity', StageName='Prospecting',
CloseDate=System.today().addMonths(1), AccountId=acct.Id);
 insert opp;
// Perform test

Test.startTest();
Database.DeleteResult result = Database.delete(acct, false);
Test.stopTest();
System.assert(!result.isSuccess());

 System.assert(result.getErrors().size() > 0);
```

```
System.assertEquals('Cannot delete account with related
opportunities.', result.getErrors()[0].getMessage());
 }
 }
```

Use Case 4:
Write a future method to count number of contacts associated account.

```
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;
    }

}


@isTest
private class AccountProcessorTest {
    @isTest
    private static void countTestContacts()
```

```
        {
            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='adam',
LastName='smith', AccountId=newAccount.id);
            insert newContact2;
        List<Id> accountIds=new List<Id>();
            accountIds.add(newAccount.Id);
            Test.startTest();
            AccountProcessor.countContacts(accountIds);
            Test.stopTest();

    }

}
```

1. What is "seeAllData"?
- It is an attribute of "@isTest" annotation.
- It grants the access of Salesforce data to "Test Class" and "Test Method".

2. Why it is recommended to avoid using "seeAllData"?

- It is recommended not to use as the code coverage of your apex class or trigger will be dependent on the data which is present in org and depending upon that the code coverage may change.

3. What is the role of Test.start() and Test.Stop()?

- Code written between these methods receives a fresh set of governor limits.

- It runs asynchronous methods synchronously.
- These can be used only once per testMethod.

4. What is System.runAs() method used for in a test Class?

- Apex code runs in system mode, where the permissions and record sharing of the current user are not taken into consideration.
- It enables us to write test methods that change the user context to an existing user or a new user so that the user record sharing is enforced.
- It doesnt enforce user permissions or field-level permissions, only record sharing.
- System.runAs() must be used only in a test method.

5. What is @testVisible used for while testing an Apex class?

- Private data members of a class are not visible outside the class.
- To make them visible to test methods, we use @testVisible before any private member in Apex class.

6. What are the best practices for test class?

- Code coverage should not depend on the existing data in the org, i.e. sellAllData should be false.
- Bulk testing should be done with atleast 200 record, to test Apex trigger and batch class.
- Testing should be done for the entire scenario not only for the code coverage.

7. What is @IsTest(OnInstall=true)?

- This annotation is used to specify which Apex tests are executed during package installation.
- This annotation is used for testing in managed or unmanaged packages.

8. What is @isTest(isParallel=true)?

- This annotation is used to indicate test classes that can run in parallel.
- @isTest(SeeAllData=true) and @isTest(isParallel=true) annotations cannot be used together on the same Apex test method.

9. Which Objects can we access without using seeAllData=true?

- User, profile, organization, AsyncApexjob, CornTrigger, RecordType, ApexClass, ApexComponent, ApexPage and custom metadata types.

10.What is @testSetup annotation?

- Methods that are annotated with @testSetup are used to create test records once and then access them in every test method in the test class.
- By setting up records once for the class, we dont need to re-create records for each test method.