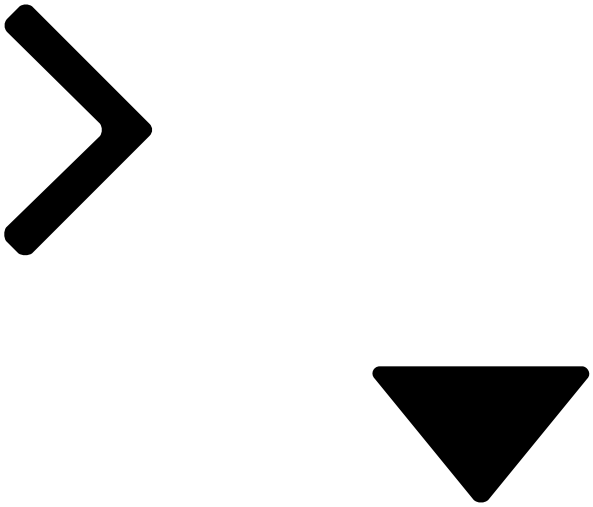
1. Apex Testing



2. Create Test Data for Apex Tests

Create Test Data for Apex Tests

Learning Objectives

After completing this unit, you'll be able to:

- Create a test utility class.
- Use a test utility method to set up test data for various test cases.
- Execute all test methods in a class.

Create Test Data for Apex Tests

Use test utility classes to add reusable methods for test data setup.

Prerequisites

Complete the prerequisites in the previous unit, Testing Apex Triggers, if you haven't done so already.

Adding a Test Utility Class

Let's refactor the previous test method by replacing test data creation with a call to a utility class method. First, you need to create the test utility class.

The `TestDataFactory` class is a special type of class—it is a public class that is annotated with `isTest` and can be accessed only from a running test. Test utility classes contain methods that can be called by test methods to perform useful tasks, such as setting up test data. Test utility classes are excluded from the org's code size limit.

To add the `TestDataFactory` class:

1. In the Developer Console, click **File** | **New** | **Apex Class**, and enter `TestDataFactory` for the class name, and then click **OK**.
2. Replace the default class body with the following.

```
@isTest
public class TestDataFactory {
    public static List<Account> createAccountsWithOpps(Integer numAccts, Integer numOppsPerAcct) {
        List<Account> accts = new List<Account>();

        for(Integer i=0;i<numAccts;i++) {
            Account a = new Account(Name='TestAccount' + i);
            accts.add(a);
        }
        insert accts;

        List<Opportunity> opps = new List<Opportunity>();
        for (Integer j=0;j<numAccts;j++) {
            Account acct = accts[j];
            // For each account just inserted, add opportunities
            for (Integer k=0;k<numOppsPerAcct;k++) {
                opps.add(new Opportunity(Name=acct.Name + ' Opportunity ' + k,
                                         StageName='Prospecting',
                                         CloseDate=System.today().addMonths(1),
                                         AccountId=acct.Id));
            }
        }
        // Insert all opportunities for all accounts.
        insert opps;

        return accts;
    }
}
```

Copy

This test utility class contains one static method, `createAccountsWithOpps()`, which accepts the number of accounts (held in the `numAccts` parameter) and the number of related opportunities to create for each account (held in the `numOppsPerAcct` parameter). The first loop in the method creates the specified number of accounts and stores them in the `accts` list variable. After the first loop, the `insert()` DML statement is called to create all accounts in the list in the database.

The second loop creates the opportunities. Because each group of opportunities are linked to one account, the outer loop iterates through accounts and contains a nested loop that creates related opportunities for the current account. The next time the nested loop is run, opportunities are added to the same list using the `add()` method. Opportunities are linked to their parent accounts using the AccountId field. The total number of all opportunities that are created is the product of the number of opportunities with the number of accounts (`numOppsPerAcct*numAccts`). Next, the `insert()` DML statement is efficiently called outside the loop to create all opportunities in the collection for all accounts in one call only.

Finally, this method returns a list of the new accounts.

Note

Even though this method doesn't return the related opportunities, you can get those records by writing a SOQL query that makes use of the existing relationship between Account and Opportunity, such as the query used in the trigger in Testing Apex Triggers.

Calling Utility Methods for Test Data Creation

Now that you've added the test utility class, modify the test class to take advantage of this class. In the `TestAccountDeletion` class, replace the block that starts with `// Test data setup` and ends with `insert opp;` with:

```
        // Test data setup
        // Create one account with one opportunity by calling a utility method
        Account[] accts = TestDataFactory.createAccountsWithOpps(1,1);
```

Copy

The array returned by the `TestDataFactory.createAccountsWithOpps(1,1)` call contains one Account sObject.

Here's the modified test method. A shorter version!

```
@isTest
private class TestAccountDeletion {
    @isTest static void TestDeleteAccountWithOneOpportunity() {
        // Test data setup
        // Create one account with one opportunity by calling a utility method
        Account[] accts = TestDataFactory.createAccountsWithOpps(1,1);

        // Perform test
        Test.startTest();
        Database.DeleteResult result = Database.delete(accts[0], false);
        Test.stopTest();
        // Verify that the deletion should have been stopped by the trigger,
        // so check that we got back an error.
        System.assert(!result.isSuccess());
        System.assert(result.getErrors().size() > 0);
        System.assertEquals('Cannot delete account with related opportunities.',
                             result.getErrors()[0].getMessage());
    }
}
```

Copy

Testing for Different Conditions

One test method is not enough to test all the possible inputs for the trigger. We need to test some other conditions, such as when an account without opportunities is deleted. We also need to test the same scenarios with a bulk number of records instead of just a single record. Here is an updated version of the test class that contains the three additional test methods. Save this updated version of the class.

```
@isTest
private class TestAccountDeletion {
    @isTest static void TestDeleteAccountWithOneOpportunity() {
        // Test data setup
        // Create one account with one opportunity by calling a utility method
        Account[] accts = TestDataFactory.createAccountsWithOpps(1,1);

        // Perform test
        Test.startTest();
        Database.DeleteResult result = Database.delete(accts[0], false);
```

```
        Test.stopTest();
        // Verify that the deletion should have been stopped by the trigger,
        // so check that we got back an error.
        System.assert(!result.isSuccess());
        System.assert(result.getErrors().size() > 0);
        System.assertEquals('Cannot delete account with related opportunities.',
                             result.getErrors()[0].getMessage());
    }

    @isTest static void TestDeleteAccountWithNoOpportunities() {
        // Test data setup
        // Create one account with no opportunities by calling a utility method
        Account[] accts = TestDataFactory.createAccountsWithOpps(1,0);

        // Perform test
        Test.startTest();
        Database.DeleteResult result = Database.delete(accts[0], false);
        Test.stopTest();
        // Verify that the deletion was successful
        System.assert(result.isSuccess());
    }

    @isTest static void TestDeleteBulkAccountsWithOneOpportunity() {
        // Test data setup
        // Create accounts with one opportunity each by calling a utility method
        Account[] accts = TestDataFactory.createAccountsWithOpps(200,1);

        // Perform test
        Test.startTest();
        Database.DeleteResult[] results = Database.delete(accts, false);
        Test.stopTest();
        // Verify for each record.
        // In this case the deletion should have been stopped by the trigger,
        // so check that we got back an error.
        for(Database.DeleteResult dr : results) {
            System.assert(!dr.isSuccess());
            System.assert(dr.getErrors().size() > 0);
            System.assertEquals('Cannot delete account with related opportunities.',
                                dr.getErrors()[0].getMessage());
        }
    }

    @isTest static void TestDeleteBulkAccountsWithNoOpportunities() {
        // Test data setup
        // Create accounts with no opportunities by calling a utility method
        Account[] accts = TestDataFactory.createAccountsWithOpps(200,0);

        // Perform test
        Test.startTest();
        Database.DeleteResult[] results = Database.delete(accts, false);
        Test.stopTest();
        // For each record, verify that the deletion was successful
        for(Database.DeleteResult dr : results) {
            System.assert(dr.isSuccess());
        }
    }
}
```

Copy

Running All Test Methods

The final step is to run the test methods in our test class, now that the class contains more comprehensive tests and has been refactored to use a test data factory. Since you've already run the tests in the `TestAccountDeletion` class, you can just rerun this test class to run all its test methods.

1. To execute the same test run, click the **Tests** tab, select your test run, and then click **Test** | **Rerun**.
2. Check the results in the Tests tab by expanding the latest test run. The test run should report that all four tests passed!

Resources

Documentation

Apex Developer Guide: Common Test Utility Classes for Test Data Creation

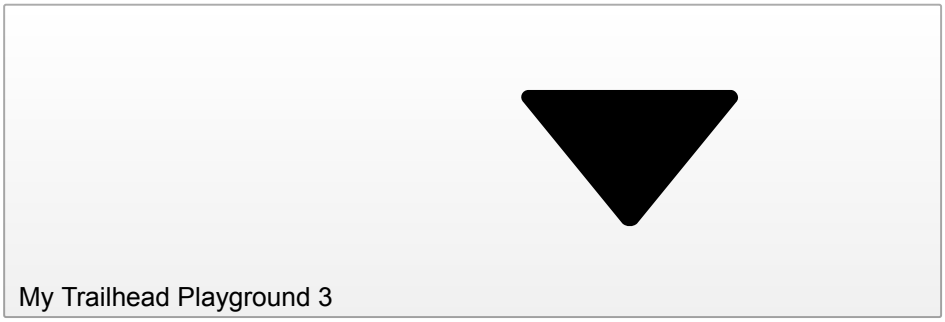
Get Ready

You'll be completing this challenge in your own personal Salesforce environment. Choose from the dropdown menu, then click **Launch** to get started. If you use Trailhead in a language other than English, set the language of your Trailhead Playground to English before you attempt this challenge. Want to find out more about using hands-on orgs for Trailhead learning? Check out the Trailhead Playground Management module.

Your Challenge

Create a contact test factory.
Create an Apex class that returns a list of contacts based on two incoming parameters: one for the number of contacts to generate, and the other for the last name. The list should NOT be inserted into the system, only returned. The first name should be dynamically generated and should be unique for each contact record in the list.

- The Apex class must be called 'RandomContactFactory' and be in the public scope.
- The Apex class should NOT use the @isTest annotation.
- The Apex class must have a public static method called 'generateRandomContacts' (without the @testMethod annotation).
- The 'generateRandomContacts' method must accept an integer as the first parameter, and a string as the second. The first parameter controls the number of contacts being generated, the second is the last name of the contacts generated.
- The 'generateRandomContacts' method should have a return type of List<Contact>.
- The 'generateRandomContacts' method must be capable of consistently generating contacts with unique first names.
- For example, the 'generateRandomContacts' might return first names based on iterated number (i.e. 'Test 1','Test 2').
- The 'generateRandomContacts' method should not insert the contact records into the database.



Launch

Check challenge to earn 500 points