Use Future Methods

**Learning Objectives**

After completing this unit, you’ll know:

* When to use future methods.
* The limitations of using future methods.
* How to use future methods for callouts.
* Future method best practices.

**Follow Along with Trail Together**

Want to follow along with an instructor as you work through this step? Take a look at this video, part of the Trail Together series on Trailhead Live.

(This clip starts at the 9:39 minute mark, in case you want to rewind and watch the beginning of the step again.)

**Future Apex**

Future Apex is used to run processes in a separate thread, at a later time when system resources become available.

Note: Technically, you use the @future annotation to identify methods that run asynchronously. However, because "methods identified with the @future annotation" is laborious, they are commonly referred to as "future methods" and that’s how we’ll reference them for the remainder of this module.

When using synchronous processing, all method calls are made from the same thread that is executing the Apex code, and no additional processing can occur until the process is complete. You can use future methods for any operation you’d like to run asynchronously in its own thread. This provides the benefits of not blocking the user from performing other operations and providing higher governor and execution limits for the process. Everyone’s a winner with asynchronous processing.

Future methods are typically used for:

* Callouts to external Web services. If you are making callouts from a trigger or after performing a DML operation, you must use a future or queueable method. A callout in a trigger would hold the database connection open for the lifetime of the callout and that is a "no-no" in a multitenant environment.
* Operations you want to run in their own thread, when time permits such as some sort of resource-intensive calculation or processing of records.
* Isolating DML operations on different sObject types to prevent the mixed DML error. This is somewhat of an edge-case but you may occasionally run across this issue. See [sObjects That Cannot Be Used Together in DML Operations](https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_dml_non_mix_sobjects.htm" \t "_blank) for more details.

**Future Method Syntax**

Future methods must be static methods, and can only return a void type. The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types. Notably, future methods can’t take standard or custom objects as arguments. A common pattern is to pass the method a List of record IDs that you want to process asynchronously.

public class SomeClass {

@future

public static void **someFutureMethod**(List<Id> recordIds) {

List<Account> accounts = [Select Id, Name from Account Where Id IN :recordIds];

// process account records to do awesome stuff

}

}

Copy

Note

The reason why objects can’t be passed as arguments to future methods is because the object can change between the time you call the method and the time that it actually executes. Remember, future methods are executed when system resources become available. In this case, the future method may have an old object value when it actually executes, which can cause all sorts of bad things to happen.

It’s important to note that future methods are not guaranteed to execute in the same order as they are called. Again, future methods are not guaranteed to execute in the same order as they are called. When using future methods, it’s also possible that two future methods could run concurrently, which could result in record locking and a nasty runtime error if the two methods were updating the same record.

**Sample Callout Code**

To make a Web service callout to an external service or API, you create an Apex class with a future method that is marked with (callout=true). The class below has methods for making the callout both synchronously and asynchronously where callouts are not permitted. We insert a record into a custom log object to track the status of the callout simply because logging is always fun to do!

public class SMSUtils {

// Call async from triggers, etc, where callouts are not permitted.

@future(callout=true)

public static void **sendSMSAsync**(String fromNbr, String toNbr, String m) {

String results = **sendSMS**(fromNbr, toNbr, m);

System.**debug**(results);

}

// Call from controllers, etc, for immediate processing

public static String **sendSMS**(String fromNbr, String toNbr, String m) {

// Calling 'send' will result in a callout

String results = SmsMessage.**send**(fromNbr, toNbr, m);

insert new SMS\_Log\_\_c(to\_\_c=toNbr, from\_\_c=fromNbr, msg\_\_c=results);

return results;

}

}

Copy

**Test Classes**

Testing future methods is a little different than typical Apex testing. To test future methods, enclose your test code between the startTest and stopTest test methods. The system collects all asynchronous calls made after the startTest. When stopTest is executed, all these collected asynchronous processes are then run synchronously. You can then assert that the asynchronous call operated properly.

Note

Test code cannot actually send callouts to external systems, so you’ll have to ‘mock’ the callout for test coverage. Check out the [Apex Integration Services](https://trailhead.salesforce.com/module/apex_integration_services) module for complete details on mocking callouts for testing.

Here’s our mock callout class used for testing. The Apex testing framework utilizes this ‘mock’ response instead of making the actual callout to the REST API endpoint.

@isTest

public class SMSCalloutMock implements HttpCalloutMock {

public HttpResponse **respond**(HttpRequest req) {

// Create a fake response

HttpResponse res = new HttpResponse();

res.**setHeader**('Content-Type**', '**application/json');

res.**setBody**('{**"status"**:**"success"**}');

res.**setStatusCode**(200);

return res;

}

}

Copy

The test class contains a single test method, which tests both the asynchronous and synchronous methods as the former calls the latter.

@IsTest

private class Test\_SMSUtils {

@IsTest

private static void **testSendSms**() {

Test.**setMock**(HttpCalloutMock.class, new SMSCalloutMock());

Test.**startTest**();

SMSUtils.**sendSMSAsync**(**'111'**, **'222'**, 'Greetings!');

Test.**stopTest**();

// runs callout and check results

List<SMS\_Log\_\_c> logs = [select msg\_\_c from SMS\_Log\_\_c];

System.**assertEquals**(1, logs.**size**());

System.**assertEquals**('success', logs[0].msg\_\_c);

}

}

Copy

**Best Practices**

Since every future method invocation adds one request to the asynchronous queue, avoid design patterns that add large numbers of future requests over a short period of time. If your design has the potential to add 2000 or more requests at a time, requests could get delayed due to flow control. Here are some best practices you want to keep in mind:

* Ensure that future methods execute as fast as possible.
* If using Web service callouts, try to bundle all callouts together from the same future method, rather than using a separate future method for each callout.
* Conduct thorough testing at scale. Test that a trigger enqueuing the @future calls is able to handle a trigger collection of 200 records. This helps determine if delays may occur given the design at current and future volumes.
* Consider using Batch Apex instead of future methods to process large number of records asynchronously. This is more efficient than creating a future request for each record.

**Things to Remember**

Future methods are a great tool, but with great power comes great responsibility. Here are some things to keep in mind when using them:

* Methods with the future annotation must be static methods, and can only return a void type.
* The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types; future methods can’t take objects as arguments.
* Future methods won’t necessarily execute in the same order they are called. In addition, it’s possible that two future methods could run concurrently, which could result in record locking if the two methods were updating the same record.
* Future methods can’t be used in Visualforce controllers in getMethodName(), setMethodName(), nor in the constructor.
* You can’t call a future method from a future method. Nor can you invoke a trigger that calls a future method while running a future method. See the link in the Resources for preventing recursive future method calls.
* The getContent() and getContentAsPDF() methods can’t be used in methods with the future annotation.
* You’re limited to 50 future calls per Apex invocation, and there’s an additional limit on the number of calls in a 24-hour period. For more information on limits, see the link below.

**Resources**

* [Future Methods](https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_invoking_future_methods.htm" \t "_blank)
* [System.isFuture() method](https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_methods_system_system.htm)
* [sObjects That Cannot Be Used Together in DML Operations](https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_dml_non_mix_sobjects.htm)
* [Execution Governors and Limits](https://developer.salesforce.com/docs/atlas.en-us.224.0.apexcode.meta/apexcode/apex_gov_limits.htm)

Top of Form

**Hands-on Challenge**

**+500 points**

**GET READY**

You’ll be completing this unit in your own hands-on org. Click **Launch** to get started, or click the name of your org to choose a different one.

If you use Trailhead in a language other than English, make sure that your hands-on org is set to the same language as the challenge instructions. Otherwise you may run into issues passing this unit. Want to find out more about using hands-on orgs on Trailhead? Check out [Trailhead Playground Management](https://trailhead.salesforce.com/en/content/learn/modules/trailhead_playground_management).

**YOUR CHALLENGE**

**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.

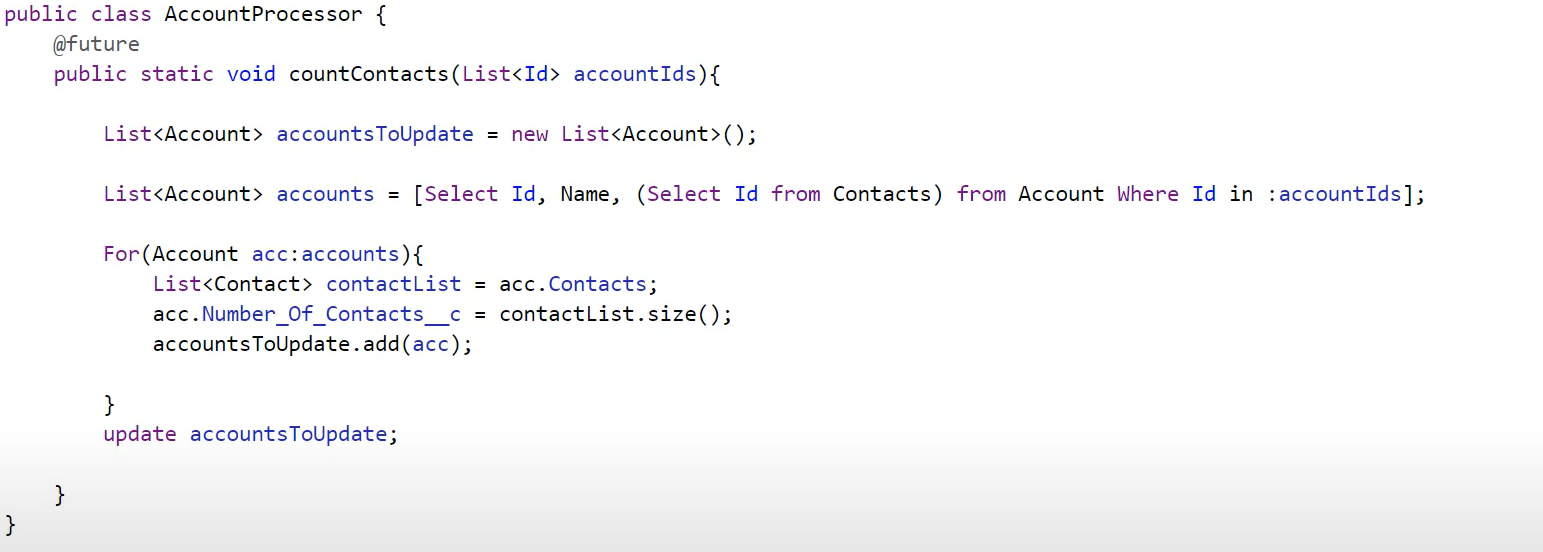
* Create a field on the Account object:
  + Label: Number Of Contacts
  + Name: Number\_Of\_Contacts
  + Type: **Number**
  + This field will hold the total number of Contacts for the Account
* Create an Apex class:
  + Name: AccountProcessor
  + Method name: countContacts
  + The method must accept 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

**swamy2**

Bottom of Form

Objectmanager🡪 Account🡪 Fields and Relation ships-🡪 New🡪 Number

First write a class:



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);

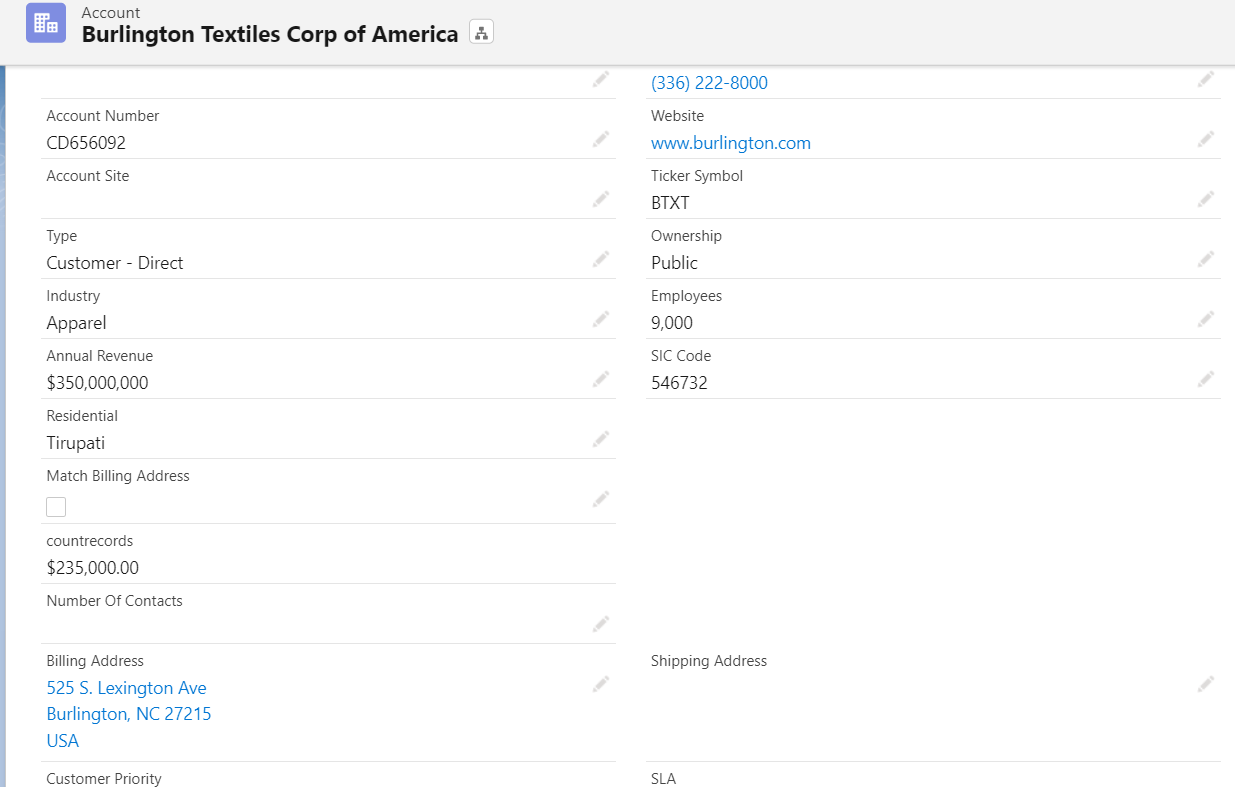
}

update accountsToUpdate;

}

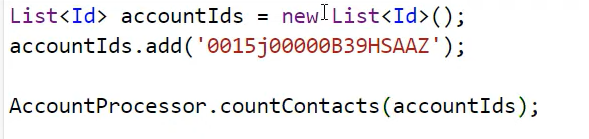
}

Open accounts-🡪



Identify Number of Contacts added here

In adebug console enter following code:



Refresh your account page then observe it will updated.

@isTest

public class AccountProcessorTest {

private static void testCountContacts(){

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='Jane',LastName='Doe',AccountId=newAccount.Id);

insert newContact2;

List<Id> accountIds=new List<Id>();

accountIds.add(newAccount.Id);

Test.startTest();

AccountProcessor.countContacts(accountIds);

Test.stopTest();

}

}

Another code:

@IsTest

public class AccountProcessorTest {

public static testmethod void TestAccountProcessorTest(){

Account a = new Account();

a.Name = 'Test Account';

Insert a;

Contact cont = New Contact();

cont.FirstName ='Bob';

cont.LastName ='Masters';

cont.AccountId = a.Id;

Insert cont;

List<Id> setAccId = new List<ID>();

setAccId.add(a.id);

Test.startTest();

AccountProcessor.countContacts(setAccId);

Test.stopTest();

}

}

