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.

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 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.

```
global 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
  }
}
```

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. If you need this type of functionality then Queueable Apex might be a better solution. 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;
    }

}
```

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]() 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
global class SMSCalloutMock implements HttpCalloutMock {
    global 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;
    }
}
```

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);
  }

}
```

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
- sObjects That Cannot Be Used Together in DML Operations
- Execution Governors and Limits
- Preventing Recursive Future Method Calls in Salesforce

Remember, this module is meant for Salesforce Classic. When you launch your hands-on org, switch to Salesforce Classic to complete this challenge.

Hands-on Challenge +500 points

You'll be completing this challenge in your own personal Salesforce environment. Ready to get hands-on?

Default TP

- Choose your hands-on org
- Default TP
-
- Log into a Developer Edition
- Create a Trailhead Playground
- Manage my hands-on orgs

LaunchHands-On Org Help

Create an Apex class that uses the @future annotation to update Account records.

Create an Apex class with a method using the @future annotation 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.

- Create a field on the Account object called 'Number_of_Contacts__c' of type Number. This field will hold the total number of Contacts for the Account.
- Create an Apex class called 'AccountProcessor' that contains a 'countContacts' method that accepts a List of Account IDs. This method must use the @future annotation.
- For each Account ID passed to the method, count the number of Contact records associated to it and update the 'Number_of_Contacts__c' field with this value.
- Create an Apex test class called 'AccountProcessorTest'.
- The unit tests must cover all lines of code included in the AccountProcessor 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.

Check challenge

Share

0 ? 21

Time Estimate

About **20** mins

Topics

Have questions about Trailhead or having problems using it?

Trailhead Forum