1. Asynchronous Apex
2. Use Batch Apex

Use Batch Apex

Learning Objectives

After completing this unit, you'll know:

- Where to use Batch Apex.
- The higher Apex limits when using batch.
- Batch Apex syntax.
- Batch Apex best practices.

Batch Apex

Batch Apex is used to run large jobs (think thousands or millions of records!) that would exceed normal processing limits. Using Batch Apex, you can process records asynchronously in batches (hence the name, "Batch Apex") to stay within platform limits. If you have a lot of records to process, for example, data cleansing or archiving, Batch Apex is probably your best solution.

Here's how Batch Apex works under the hood. Let's say you want to process 1 million records using Batch Apex. The execution logic of the batch class is called once for each batch of records you are processing. Each time you invoke a batch class, the job is placed on the Apex job queue and is executed as a discrete transaction. This functionality has two awesome advantages:

- Every transaction starts with a new set of governor limits, making it easier to ensure that your code stays within the governor execution limits.
- If one batch fails to process successfully, all other successful batch transactions aren't rolled back.

Batch Apex Syntax

To write a Batch Apex class, your class must implement the `Database.Batchable` interface and include the following three methods:

start

Used to collect the records or objects to be passed to the interface method `execute` for processing. This method is called once at the beginning of a Batch Apex job and returns either a Database.QueryLocator object or an Iterable that contains the records or objects passed to the job.

Most of the time a `QueryLocator` does the trick with a simple SOQL query to generate the scope of objects in the batch job. But if you need to do something crazy like loop through the results of an API call or pre-process records before being passed to the `execute` method, you might want to check out the Custom Iterators link in the Resources section.

With the QueryLocator object, the governor limit for the total number of records retrieved by SOQL queries is bypassed and you can query up to 50 million records. However, with an Iterable, the governor limit for the total number of records retrieved by SOQL queries is still enforced.

execute

Performs the actual processing for each chunk or "batch" of data passed to the method. The default batch size is 200 records. Batches of records are not guaranteed to execute in the order they are received from the `start` method.

This method takes the following:

- A reference to the `Database.BatchableContext` object.
- A list of sObjects, such as `List<sObject>`, or a list of parameterized types. If you are using a `Database.QueryLocator`, use the returned list.

finish
Used to execute post-processing operations (for example, sending an email) and is called once after all batches are processed.

Here's what the skeleton of a Batch Apex class looks like:

```
global class MyBatchClass implements Database.Batchable<sObject> {
    global (Database.QueryLocator | Iterable<sObject>) start(Database.BatchableContext bc) {
        // collect the batches of records or objects to be passed to execute
    }
    global void execute(Database.BatchableContext bc, List<P> records){
        // process each batch of records
    }
    global void finish(Database.BatchableContext bc){
        // execute any post-processing operations
    }
}
```

Copy

Invoking a Batch Class

To invoke a batch class, simply instantiate it and then call `Database.executeBatch` with the instance:

```
MyBatchClass myBatchObject = new MyBatchClass();
Id batchId = Database.executeBatch(myBatchObject);
```

Copy

You can also optionally pass a second `scope` parameter to specify the number of records that should be passed into the execute method for each batch. Pro tip: you might want to limit this batch size if you are running into governor limits.

```
Id batchId = Database.executeBatch(myBatchObject, 100);
```

Copy

Each batch Apex invocation creates an AsyncApexJob record so that you can track the job's progress. You can view the progress via SOQL or manage your job in the Apex Job Queue. We'll talk about the Job Queue shortly.

```
AsyncApexJob job = [SELECT Id, Status, JobItemsProcessed, TotalJobItems, NumberOfErrors FROM AsyncApexJob WHERE ID = :batchId ];
```

Copy

Using State in Batch Apex

Batch Apex is typically stateless. Each execution of a batch Apex job is considered a discrete transaction. For example, a batch Apex job that contains 1,000 records and uses the default batch size is considered five transactions of 200 records each.

If you specify `Database.Stateful` in the class definition, you can maintain state across all transactions. When using `Database.Stateful`, only instance member variables retain their values between transactions. Maintaining state is useful for counting or summarizing records as they're processed. In our next example, we'll be updating contact records in our batch job and want to keep track of the total records affected so we can include it in the notification email.

Sample Batch Apex Code

Now that you know how to write a Batch Apex class, let's see a practical example. Let's say you have a business requirement that states that all contacts for companies in the USA must have their parent company's billing address as their mailing address. Unfortunately, users are entering new contacts without the correct addresses! Will users never learn?! Write a Batch Apex class that ensures that this requirement is enforced.

The following sample class finds all account records that are passed in by the `start()` method using a `QueryLocator` and updates the associated contacts with their account's mailing address. Finally, it sends off an email with the results of the bulk job and, since we are using `Database.Stateful` to track state, the number of records updated.

```
global class UpdateContactAddresses implements
    Database.Batchable<sObject>, Database.Stateful {

    // instance member to retain state across transactions
    global Integer recordsProcessed = 0;
    global Database.QueryLocator start(Database.BatchableContext bc) {
        return Database.getQueryLocator(
            'SELECT ID, BillingStreet, BillingCity, BillingState, ' +
            'BillingPostalCode, (SELECT ID, MailingStreet, MailingCity, ' +
            'MailingState, MailingPostalCode FROM Contacts) FROM Account ' +
            'Where BillingCountry = \'USA\''
        );
    }
    global void execute(Database.BatchableContext bc, List<Account> scope){
        // process each batch of records
        List<Contact> contacts = new List<Contact>();
        for (Account account : scope) {
            for (Contact contact : account.contacts) {
                contact.MailingStreet = account.BillingStreet;
                contact.MailingCity = account.BillingCity;
                contact.MailingState = account.BillingState;
                contact.MailingPostalCode = account.BillingPostalCode;
                // add contact to list to be updated
                contacts.add(contact);
                // increment the instance member counter
                recordsProcessed = recordsProcessed + 1;
            }
        }
        update contacts;
    }
    global void finish(Database.BatchableContext bc){
        System.debug(recordsProcessed + ' records processed. Shazam!');
        AsyncApexJob job = [SELECT Id, Status, NumberOfErrors,
            JobItemsProcessed,
            TotalJobItems, CreatedBy.Email
            FROM AsyncApexJob
            WHERE Id = :bc.getJobId()];
        // call some utility to send email
        EmailUtils.sendMessage(job, recordsProcessed);
    }
}
```

Copy

The code should be fairly straightforward but can be a little abstract in reality. Here's what's going on in more detail:

- The `start` method provides the collection of all records that the `execute` method will process in individual batches. It returns the list of records to be processed by calling `Database.getQueryLocator` with a SOQL query. In this case we are simply querying for all Account records with a Billing Country of 'USA'.
- Each batch of 200 records is passed in the second parameter of the `execute` method. The `execute` method sets each contact's mailing address to the accounts' billing address and increments `recordsProcessed` to track the number of records processed.
- When the job is complete, the `finish` method performs a query on the AsyncApexJob object (a table that lists information about batch jobs) to get the status of the job, the submitter's email address, and some other information. It then sends a notification email to the job submitter that includes the job info and number of contacts updated.

Testing Batch Apex

Since Apex development and testing go hand in hand, here’s how we test the above batch class. In a nutshell, we insert some records, call the Batch Apex class and then assert that the records were updated properly with the correct address.

```
@isTest
private class UpdateContactAddressesTest {
    @testSetup
    static void setup() {
        List<Account> accounts = new List<Account>();
        List<Contact> contacts = new List<Contact>();
        // insert 10 accounts
        for (Integer i=0;i<10;i++) {
            accounts.add(new Account(name='Account '+i,
                billingcity='New York', billingcountry='USA'));
        }
        insert accounts;
        // find the account just inserted. add contact for each
        for (Account account : [select id from account]) {
            contacts.add(new Contact(firstname='first',
                lastname='last', accountId=account.id));
        }
        insert contacts;
    }
    static testmethod void test() {
        Test.startTest();
        UpdateContactAddresses uca = new UpdateContactAddresses();
        Id batchId = Database.executeBatch(uca);
        Test.stopTest();
        // after the testing stops, assert records were updated properly
        System.assertEquals(10, [select count() from contact where MailingCity = 'New York']);
    }

}
```

Copy

The `setup` method inserts 10 account records with the billing city of ‘New York’ and the billing country of ‘USA’. Then for each account, it creates an associated contact record. This data is used by the batch class.

Note

Make sure that the number of records inserted is less than the batch size of 200 because test methods can execute only one batch total.

In the test method, the `UpdateContactAddresses` batch class is instantiated, invoked by calling `Database.executeBatch` and passing it the instance of the batch class.

The call to `Database.executeBatch` is included within the `Test.startTest` and `Test.stopTest` block. This is where all of the magic happens. The job executes after the call to `Test.stopTest`. Any asynchronous code included within `Test.startTest` and `Test.stopTest` is executed synchronously after `Test.stopTest`.

Finally, the test verifies that all contact records were updated correctly by checking that the number of contact records with the billing city of ‘New York’ matches the number of records inserted (i.e., 10).

Best Practices

As with future methods, there are a few things you want to keep in mind when using Batch Apex. To ensure fast execution of batch jobs, minimize Web service callout times and tune queries used in your batch Apex code. The longer the batch job executes, the more likely other queued jobs are delayed when many jobs are in the queue. Best practices include:

- Only use Batch Apex if you have more than one batch of records. If you don't have enough records to run more than one batch, you are probably better off using Queueable Apex.
- Tune any SOQL query to gather the records to execute as quickly as possible.
- Minimize the number of asynchronous requests created to minimize the chance of delays.

- Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger won't add more batch jobs than the limit.

Resources

- Batch Apex
- Using Batch Apex
- Custom Iterators

Note

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

Assessment Complete!

+500 points

Asynchronous Apex
100%
Progress: 100%
Retake this Challenge
View more modules