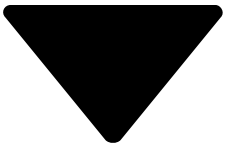
1. Asynchronous Apex



2. Control Processes with Queueable Apex

Control Processes with Queueable Apex

Learning Objectives

After completing this unit, you'll know:

- When to use the Queueable interface.
- The differences between queueable and future methods.
- Queueable Apex syntax.
- Queueable method best practices.

Queueable Apex

Released in Winter '15, Queueable Apex is essentially a superset of future methods with some extra #awesomesauce. We took the simplicity of future methods and the power of Batch Apex and mixed them together to form Queueable Apex! It gives you a class structure that the platform serializes for you, a simplified interface without `start` and `finish` methods and even allows you to utilize more than just primitive arguments! It is called by a simple `System.enqueueJob()` method, which returns a job ID that you can monitor. It beats sliced bread hands down!

Queueable Apex allows you to submit jobs for asynchronous processing similar to future methods with the following additional benefits:

- Non-primitive types: Your Queueable class can contain member variables of non-primitive data types, such as sObjects or custom Apex types. Those objects can be accessed when the job executes.
- Monitoring: When you submit your job by invoking the `System.enqueueJob` method, the method returns the ID of the `AsyncApexJob` record. You can use this ID to identify your job and monitor its progress, either through the Salesforce user interface in the Apex Jobs page, or programmatically by querying your record from `AsyncApexJob`.
- Chaining jobs: You can chain one job to another job by starting a second job from a running job. Chaining jobs is useful if you need to do some sequential processing.

Queueable Versus Future

Because queueable methods are functionally equivalent to future methods, most of the time you'll probably want to use queueable instead of future methods. However, this doesn't necessarily mean you should go back and refactor all your future methods right now.

Another reason to use future methods instead of queueable is when your functionality is sometimes executed synchronously, and sometimes asynchronously. It's much easier to refactor a method in this manner than converting to a queueable class. This is handy when you discover that part of your existing code needs to be moved to async execution. You can simply create a similar future method that wraps your synchronous method like so:

```
@future
static void myFutureMethod(List<String> params) {
    // call synchronous method
    mySyncMethod(params);
}
```

Copy

Queueable Syntax

To use Queueable Apex, simply implement the Queueable interface.

```
public class SomeClass implements Queueable {
    public void execute(QueueableContext context) {
        // awesome code here
    }
}
```

Copy

Sample Code

A common scenario is to take some set of sObject records, execute some processing such as making a callout to an external REST endpoint or perform some calculations and then update them in the database asynchronously. Since `@future` methods are limited to primitive data types (or arrays or collections of primitives), queueable Apex is an ideal choice. The following code takes a collection of Account records, sets the parentId for each record, and then updates the records in the database.

```
public class UpdateParentAccount implements Queueable {

    private List<Account> accounts;
    private ID parent;

    public UpdateParentAccount(List<Account> records, ID id) {
        this.accounts = records;
        this.parent = id;
    }
    public void execute(QueueableContext context) {
        for (Account account : accounts) {
          account.parentId = parent;
          // perform other processing or callout
        }
        update accounts;
    }
```

```
}
```

Copy

To add this class as a job on the queue, execute the following code:

```
// find all accounts in 'NY'
List<Account> accounts = [select id from account where billingstate = 'NY'];
// find a specific parent account for all records
Id parentId = [select id from account where name = 'ACME Corp'][0].Id;
// instantiate a new instance of the Queueable class
UpdateParentAccount updateJob = new UpdateParentAccount(accounts, parentId);
// enqueue the job for processing
ID jobID = System.enqueueJob(updateJob);
```

Copy

After you submit your queueable class for execution, the job is added to the queue and will be processed when system resources become available.

You can use the new job ID to monitor progress, either through the Apex Jobs page or programmatically by querying `AsyncApexJob`:

```
SELECT Id, Status, NumberOfErrors FROM AsyncApexJob WHERE Id = :jobID
```

Copy

Testing Queueable Apex

The following code sample shows how to test the execution of a queueable job in a test method. It looks very similar to Batch Apex testing. To ensure that the queueable process runs within the test method, the job is submitted to the queue between the `Test.startTest` and `Test.stopTest` block. The system executes all asynchronous processes started in a test method synchronously after the `Test.stopTest` statement. Next, the test method verifies the results of the queueable job by querying the account records that the job updated.

```
@isTest
public class UpdateParentAccountTest {
    @testSetup
    static void setup() {
        List<Account> accounts = new List<Account>();
        // add a parent account
        accounts.add(new Account(name='Parent'));
        // add 100 child accounts
        for (Integer i = 0; i < 100; i++) {
            accounts.add(new Account(
                name='Test Account'+i
            ));
        }
        insert accounts;
    }

    static testmethod void testQueueable() {
        // query for test data to pass to queueable class
        Id parentId = [select id from account where name = 'Parent'][0].Id;
        List<Account> accounts = [select id, name from account where name like 'Test Account%'];
        // Create our Queueable instance
        UpdateParentAccount updater = new UpdateParentAccount(accounts, parentId);
        // startTest/stopTest block to force async processes to run
```

```
        Test.startTest();
        System.enqueueJob(updater);
        Test.stopTest();
        // Validate the job ran. Check if record have correct parentId now
        System.assertEquals(100, [select count() from account where parentId = :parentId]);
    }

}
```

Copy

Chaining Jobs

One of the best features of Queueable Apex is job chaining. If you ever need to run jobs sequentially, Queueable Apex could make your life much easier. To chain a job to another job, submit the second job from the `execute()` method of your queueable class. You can add only one job from an executing job, which means that only one child job can exist for each parent job. For example, if you have a second class called SecondJob that implements the Queueable interface, you can add this class to the queue in the `execute()` method as follows:

```
public class FirstJob implements Queueable {
    public void execute(QueueableContext context) {
        // Awesome processing logic here
        // Chain this job to next job by submitting the next job
        System.enqueueJob(new SecondJob());
    }
}
```

Copy

Once again, testing has a slightly different pattern. You can't chain queueable jobs in an Apex test, doing so results in an error. To avoid nasty errors, you can check if Apex is running in test context by calling `Test.isRunningTest()` before chaining jobs.

Things to Remember

Queueable Apex is a great new tool but there are a few things to watch out for:

- The execution of a queued job counts once against the shared limit for asynchronous Apex method executions.
- You can add up to 50 jobs to the queue with `System.enqueueJob` in a single transaction.
- When chaining jobs, you can add only one job from an executing job with `System.enqueueJob`, which means that only one child job can exist for each parent queueable job. Starting multiple child jobs from the same queueable job is a no-no.
- No limit is enforced on the depth of chained jobs, which means that you can chain one job to another job and repeat this process with each new child job to link it to a new child job. However, for Developer Edition and Trial orgs, the maximum stack depth for chained jobs is 5, which means that you can chain jobs four times and the maximum number of jobs in the chain is 5, including the initial parent queueable job.

Resources

- Queueable Apex

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