Salesforce Bootcamp Training Program

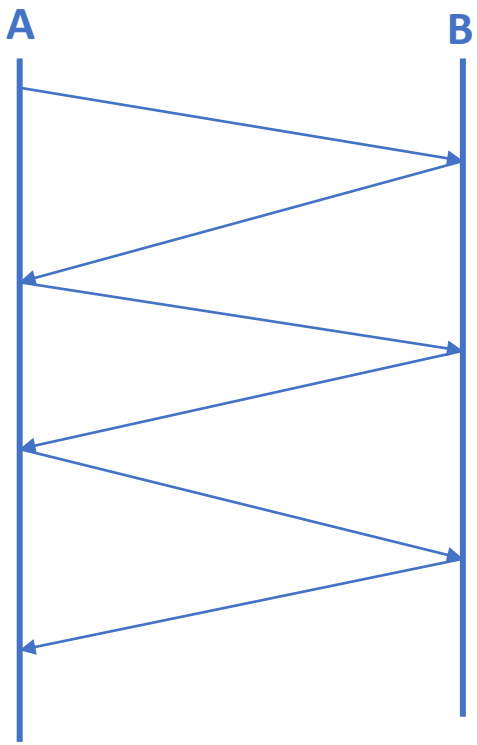
Deloitte. Digital

Batch APEX

6th April, 2022

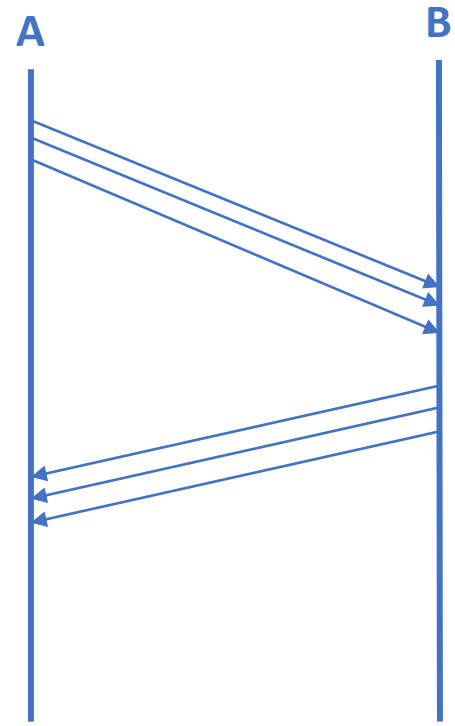
Synchronous Vs Asynchronous

SYNCHRONOUS



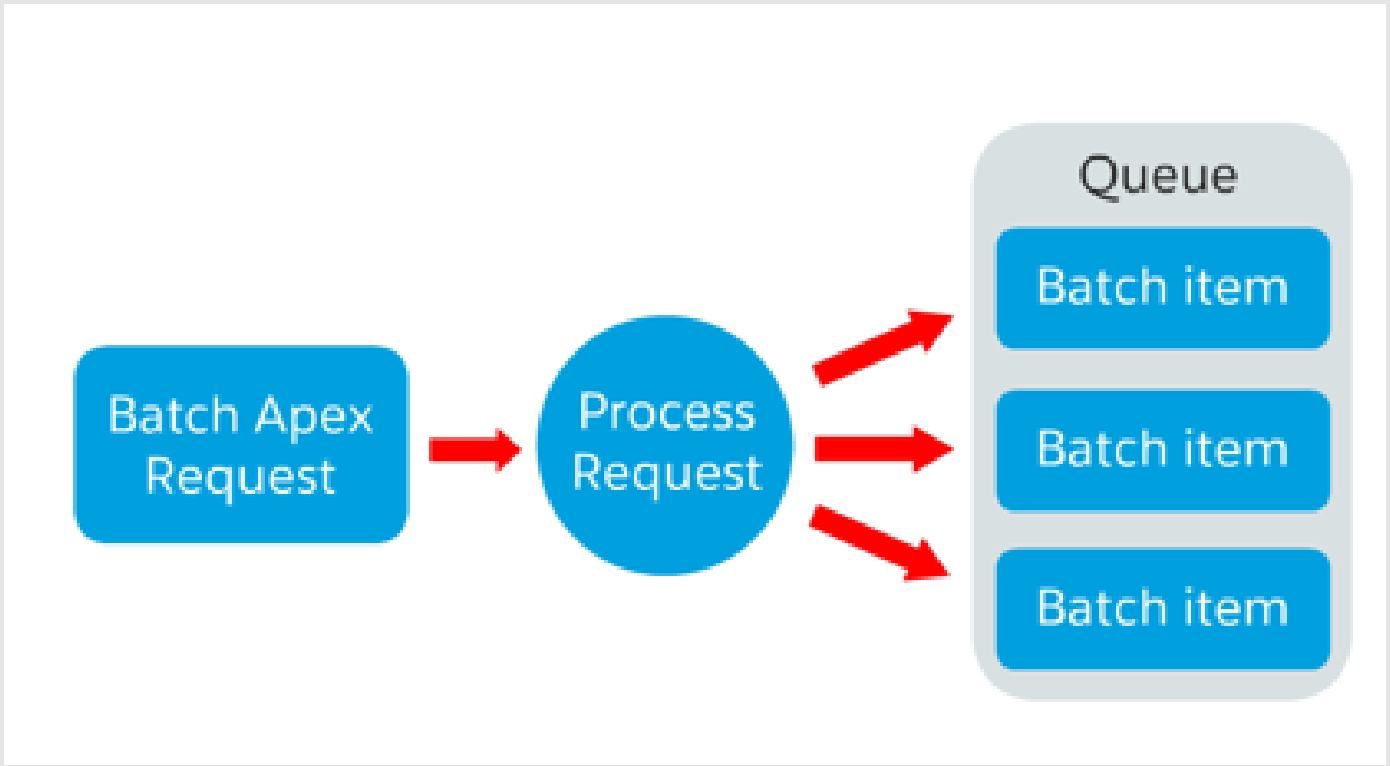
When you execute something synchronously, you wait for the previous task to get over before starting the next task

ASYNCHRONOUS



When you execute something asynchronously, you **don't** wait for the previous task to get over before starting the next task

What is Batch Apex?



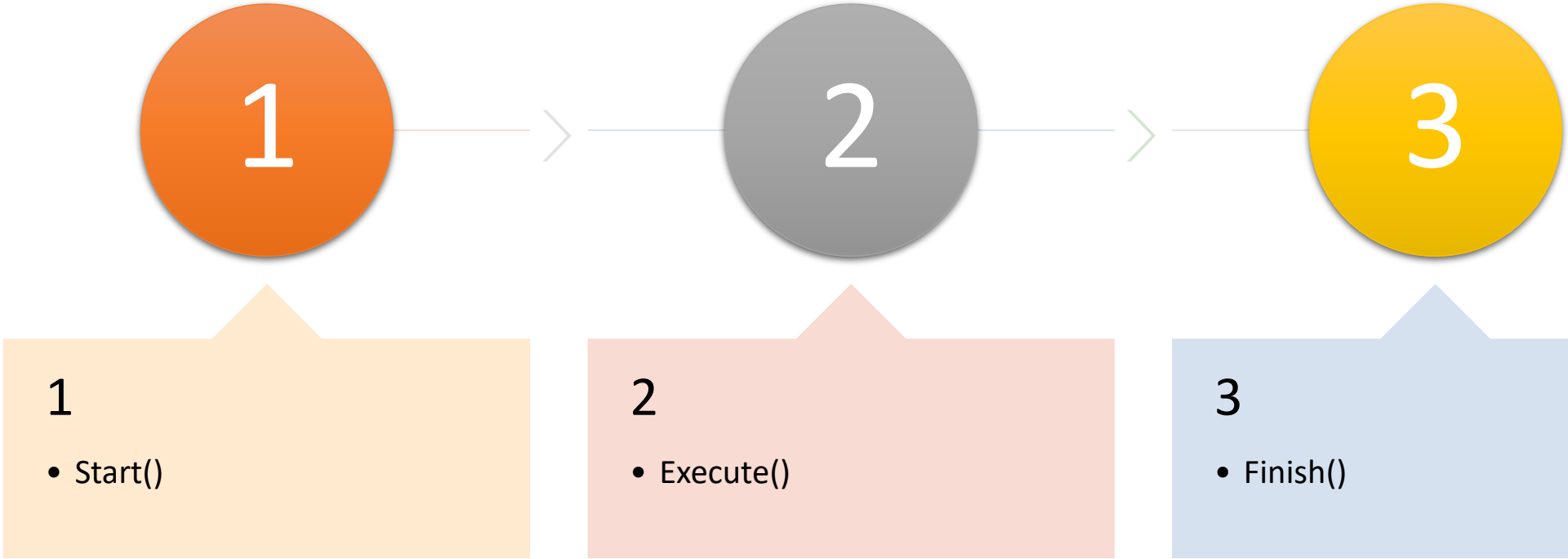
Batch class in salesforce is used to run large jobs (think thousands or millions of records!) that would exceed normal processing limits.

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.

Using Batch Apex

- When we are using the Batch Apex, we must implement the Salesforce-provided interface **Database.Batchable**, and then invoke the class programmatically.
- Database.Batchable interface has three methods which we must implement as given below:



Start()

start()

execute()

finish()

- The start method is called at the beginning of a batch Apex job.
- You use the start method to collect records or objects to be passed to the execute method.
- This method returns either a ***Database.getQueryLocator*** object or an ***Iterable*** that contains the records or objects being passed into the job.
- *Syntax*

```
global (Database.QueryLocator | Iterable) start(Database.BatchableContext bc) {}
```

- Note the following points:
  - Use the ***Database.QueryLocator*** object when you are using a simple query to generate the scope of objects used in the batch job. In this case, the SOQL data row limit will be bypassed.
  - Use ***Iterable*** object when you have complex criteria to process the records. Database.QueryLocator determines the scope of records which should be processed.

execute()

start()

execute()

finish()

- The execute method is called for each batch item of records passed to the method.
- You use this method to perform all required processing for each chunk of data.
- Each execution of a batch item is considered a discrete transaction, and the Apex governor limits are reset.
- The execute method takes an optional scope parameter, which can be used to limit the number of records to be passed in each batch item to lower than the **200** records default

```
global void  execute(Database.BatchableContext BC , List<sObject>) {}
```


finish()

start()

execute()

finish()

- The finish method is called after all batches are processed.
- You use this method to send confirmation emails or execute post-processing operations.
- This method is called once all the batches are executed

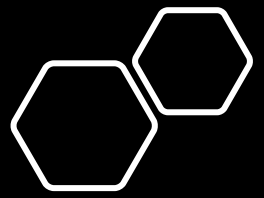
```
global void  finish(Database.BatchableContext BC) {}
```


How does the code look 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<sObject> records){
        // process each batch of records
    }

    global void finish(Database.BatchableContext bc){
        // execute any post-processing operations
    }
}
```

Batch Class Example

Fetch all the Account records and update their name by adding "bootcamp" at the end of their name

```
global class BatchApexExample implements Database.Batchable<sObject> {
   global Database.QueryLocator start(Database.BatchableContext BC) {
      // collect the batches of records or objects to be passed to execute

      String query = 'SELECT Id, Name FROM Account';
      return Database.getQueryLocator(query);
   }
   global void execute(Database.BatchableContext BC, List<Account> accList) {
      // process each batch of records default size is 200
      for(Account acc : accList) {
         // Update the Account Name
         acc.Name = acc.Name + ' bootcamp';
      }
      try {
         // Update the Account Record
         update accList;
      } catch(Exception e) {
         System.debug(e);
      }
   }
   global void finish(Database.BatchableContext BC) {
      // execute any post-processing operations like sending email
   }
}
```


Invoking Batch Class

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

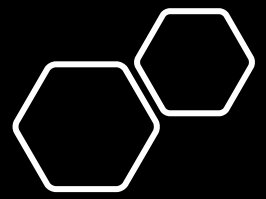
```
BatchApexExample objBatchApexExample = new BatchApexExample();
Id batchId = Database.executeBatch(objBatchApexExample);
```

- 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. (You might want to limit this batch size if you are running into governor limits.)

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

- 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.

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

Scheduling Batch Apex

- You can also use the Schedulable interface with batch Apex classes.
- The following example implements the Schedulable interface for a batch Apex class:

```
global class scheduledBatchable implements Schedulable {

  global void execute(SchedulableContext sc) {

    BatchApexExample b = new BatchApexExample();
    Database.executeBatch(b);
  }
}
```


Things to remember

*Up to **5** batch jobs can be queued or active concurrently.*

*The maximum number of batch Apex method executions per 24-hour period is 250,000, or the number of user licenses in your org multiplied by **200**—whichever is greater.*

*A maximum of **50 million** records can be returned in the QueryLocator object. If more than 50 million records are returned, the batch job is immediately terminated and marked as Failed.*

*If the start method of the batch class returns a QueryLocator, the optional scope parameter of executeBatch can have a maximum value of **2,000**.*

*If the start method of the batch class returns an iterable, the scope parameter value has **no upper limit**.*

*The start, execute, and finish methods can implement up to **100** callouts each. Implement **AllowsCallouts** for enabling callouts from the Batch apex.*

*Methods declared as **future** can't be called from a batch Apex class.*

*All methods in the class must be defined as **global** or **public**.*

Deloitte.
Digital

Thank You!

