Batch Class in Apex – Salesforce

A **Batch Apex class** is used to process large volumes of records asynchronously in Salesforce. It allows data processing in chunks (or batches), making it more efficient than standard synchronous operations.

## **Key Components of a Batch Apex Class**

A Batch Apex class in Salesforce must implement the Database.Batchable<T> interface, which requires three methods:

1. **start** → Collects the records to be processed.
2. **execute** → Processes each batch of records.
3. **finish** → Performs any post-processing tasks.

## **Structure of a Batch Class**

A typical Batch Apex class follows this structure:

apex

CopyEdit

global class MyBatchClass implements Database.Batchable<sObject> {

global Database.QueryLocator start(Database.BatchableContext BC) {

// Query records to be processed

return Database.getQueryLocator('SELECT Id, Name FROM Account');

}

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

// Process each batch of records

for (Account acc : (List<Account>) scope) {

acc.Name = acc.Name + ' - Updated';

}

update scope;

}

global void finish(Database.BatchableContext BC) {

// Perform any post-processing actions

System.debug('Batch Process Completed');

}

}

## **Explanation – Word by Word**

### **1. Class Declaration**

apex

CopyEdit

global class MyBatchClass implements Database.Batchable<sObject>

* global → The class must be declared as **global** to be accessible across the system.
* class MyBatchClass → Declares a class named **MyBatchClass**.
* implements Database.Batchable<sObject> → The class implements the **Batchable** interface, which allows it to run as a batch.
* <sObject> → Indicates that this batch class will work with any standard/custom Salesforce object.

### **2. start Method**

apex

CopyEdit

global Database.QueryLocator start(Database.BatchableContext BC) {

return Database.getQueryLocator('SELECT Id, Name FROM Account');

}

* global → This method must be **global** to be accessible.
* Database.QueryLocator start(...) → Returns the records that will be processed in batches.
* Database.BatchableContext BC → Provides information about the current batch execution.
* return Database.getQueryLocator(...) → Uses **SOQL Query** to retrieve records from the database.

### **3. execute Method**

apex

CopyEdit

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

for (Account acc : (List<Account>) scope) {

acc.Name = acc.Name + ' - Updated';

}

update scope;

}

* global void → This method does **not return anything** (void).
* execute(Database.BatchableContext BC, List<sObject> scope)
  + BC → Batch context.
  + scope → A list of records from the query in start.
* for (Account acc : (List<Account>) scope) → Loops through each batch of records.
* acc.Name = acc.Name + ' - Updated' → Modifies each Account's **Name**.
* update scope; → Saves the changes to the database.

### **4. finish Method**

apex

CopyEdit

global void finish(Database.BatchableContext BC) {

System.debug('Batch Process Completed');

}

* global void → No return value.
* finish(Database.BatchableContext BC) → Runs after all batches are processed.
* System.debug('Batch Process Completed'); → Logs a message indicating completion.

## **How to Execute a Batch Class?**

To run a batch class, execute the following in **Developer Console:**

apex

CopyEdit

MyBatchClass myBatch = new MyBatchClass();

Database.executeBatch(myBatch, 200);

* new MyBatchClass() → Creates an instance of the batch class.
* Database.executeBatch(myBatch, 200);
  + Executes the batch.
  + 200 → Specifies the batch size (number of records processed per batch).

## **Advantages of Batch Apex**

✅ Processes **large datasets** asynchronously.  
✅ Reduces **governor limits** by handling data in chunks.  
✅ Can be scheduled to run **at specific times**.  
✅ **Error handling** is easier since failures only affect specific batches.

## **Limitations of Batch Apex**

❌ Maximum **50 million records** can be processed in a single batch.  
❌ Cannot make **more than one callout** per execute method.  
❌ Limited to **5 batch jobs at a time** (active and queued).

### **Stateful vs. Stateless in Batch Apex**

In **Batch Apex**, "state" refers to **the ability to retain variable values between different batch executions**. This is controlled by implementing either:

1. **Database.Stateful (Stateful Batch) → Retains state between batches**
2. **Stateless Batch (Default) → Does NOT retain state between batches**

## **1. What is State in Batch Apex?**

In Apex, each batch execution runs **independently** of the others. By default, instance variables (class-level variables) **do not retain their values** between batch executions. However, if we implement Database.Stateful, those variables will **persist across batch executions**.

### **Example Scenario**

Suppose you want to calculate the **total number of processed records** across all batches.

* Without Database.Stateful → The count resets for each batch.
* With Database.Stateful → The count is retained across batches.

## **2. Stateless Batch (Default Behavior)**

A batch class **without** Database.Stateful does **not retain values** of instance variables across batch executions.

### **Example (Stateless Batch)**

apex

CopyEdit

global class MyStatelessBatch implements Database.Batchable<sObject> {

public Integer recordCount = 0; // This variable resets for every batch

global Database.QueryLocator start(Database.BatchableContext BC) {

return Database.getQueryLocator('SELECT Id FROM Account');

}

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

recordCount += scope.size; // Count records in this batch

System.debug('Current Batch Count: ' + recordCount);

}

global void finish(Database.BatchableContext BC) {

System.debug('Final Record Count: ' + recordCount); // Will print 0 or last batch's count only

}

}

### **Expected Behavior**

* recordCount resets for each batch execution.
* At the end (finish), it does **not** give the total processed records.

## **3. Stateful Batch (Database.Stateful)**

If we need to **retain values** between batch executions, we implement Database.Stateful.

### **Example (Stateful Batch)**

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

public Integer recordCount = 0; // This variable will persist across batches

global Database.QueryLocator start(Database.BatchableContext BC) {

return Database.getQueryLocator('SELECT Id FROM Account');

}

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

recordCount += scope.size; // Accumulates across batches

System.debug('Current Batch Count: ' + recordCount);

}

global void finish(Database.BatchableContext BC) {

System.debug('Final Record Count: ' + recordCount); // Will print the total count of all batches

}

}

### **Expected Behavior**

* recordCount retains values between batches.
* At the end (finish), it will show **the total processed records across all batches**.

## **4. Key Differences: Stateful vs. Stateless**

| **Feature** | **Stateless (Default)** | **Stateful (Database.Stateful)** |
| --- | --- | --- |
| **Variable Retention** | No, resets for each batch | Yes, persists across batches |
| **Use Case** | Simple record processing | Accumulating data (e.g., total count, list storage) |
| **Governor Limits Impact** | Lower memory usage | Higher memory usage (stores state) |
| **Performance** | Faster | Slightly slower due to memory retention |
| **Example Use Cases** | Updating records, deleting records | Counting records, storing aggregate data, logging results |

## **5. When to Use Stateful vs. Stateless?**

### **✅ Use Database.Stateful When:**

✔️ You need to **store cumulative data** across batches (e.g., total records processed).  
✔️ You need to **maintain an object list** to perform operations at the end.  
✔️ You want to **log data persistently** across batches.

### **❌ Avoid Database.Stateful When:**

❌ You only perform independent updates on records.  
❌ You need to optimize **performance and memory usage**.  
❌ Your use case does **not require data retention** between batches.

## **Conclusion**

* By **default**, Batch Apex is **stateless**, meaning instance variables reset after each batch execution.
* If you implement Database.Stateful, **instance variables retain their values across batches**.
* **Use Stateful only when necessary**, as it increases **memory consumption**.