Project Title – “Smart Subscription Tracker”

Phase 5: Apex Programming (Developer)

Classes & Objects

- Purpose of the Class

The SubscriptionHelper class is designed to automatically manage subscription statuses based on their End_Date__c.

It ensures consistency in subscription records by assigning either ‘Active’ or ‘Expired’ status.

- Business Logic Implemented

If End_Date__c is in the past, the subscription is marked as Expired.

If End_Date__c is in the future or blank, the subscription is set as Active.

Subscriptions previously marked as ‘Expired’ but with a valid future date are reactivated as Active.

SubscriptionHelper.apxc | Log executeAnonymous @9/20/2025, 2:54:07 PM

Code Coverage: None | API Version: 64

```java
public with sharing class SubscriptionHelper {
    /**

     * - If End_Date__c is in the past -> 'Expired'
     * - Otherwise -> 'Active' (if blank or previously 'Expired')
     */
    public static void setStatusBeforeSave(List<Subscription_c__c> subs) {
        if (subs == null || subs.isEmpty()) return;
        Date today = Date.today();
        for (Subscription_c__c s : subs) {
            // skip if no end date
            if (s.End_Date__c == null) continue;

            if (s.End_Date__c < today) {
                s.Status__c = 'Expired';
            } else {
                // set Active if empty or was Expired
                if (String.isBlank(s.Status__c) || s.Status__c == 'Expired') {
                    s.Status__c = 'Active';
                }
            }
        }
    }
}
```

Anonyamous code window-

```
Account acc = new Account(Name = 'Test Account for Subscription');
insert acc;
// Create some test subscriptions
List<Subscription_c__c> subs = new List<Subscription_c__c>();

// One expired (yesterday)
subs.add(new Subscription_c__c(
    Name = 'Test Expired',
        Account__c = acc.Id,
    Start_Date__c = Date.today().addDays(-10),
    End_Date__c = Date.today().addDays(-1)
));

// One active (10 days later)
subs.add(new Subscription_c__c(
    Name = 'Test Active',
     Account__c = acc.Id,
    Start_Date__c = Date.today(),
    End_Date__c = Date.today().addDays(10)
));

// One with no End Date
subs.add(new Subscription_c__c(
    Name = 'No End Date',
     Account__c = acc.Id,
    Start_Date__c = Date.today()
));

// Insert records
insert subs;

// Query them back
List<Subscription_c__c> insertedSubs = [
    SELECT Id, Name, End_Date__c, Status__c
    FROM Subscription_c__c
    WHERE Id IN :subs
];

// Call the helper directly (simulating trigger)
SubscriptionHelper.setStatusBeforeSave(insertedSubs);

// Debug the results
for (Subscription_c__c s : insertedSubs) {
    System.debug('Subscription: ' + s.Name +
                 ' | End Date: ' + s.End_Date__c +
                 ' | Status: ' + s.Status__c);
}
```

Output-

File ▾ Edit ▾ Debug ▾ Test ▾ Workspace ▾ Help ▾ < >

SubscriptionHelper.apxc | Log executeAnonymous @9/20/2025, 2:54:07 PM | **Log executeAnonymous @9/20/2025, 3:58:15 PM**

Execution Log

Timestamp	Event	Details
15:58:15:324	USER_DEBUG	[44]|DEBUG|Subscription: Test Expired | End Date: 2025-09-19 00:00:00 | Status: Expired
15:58:15:324	USER_DEBUG	[44]|DEBUG|Subscription: Test Active | End Date: 2025-09-30 00:00:00 | Status: Draft
15:58:15:325	USER_DEBUG	[44]|DEBUG|Subscription: No End Date | End Date: null | Status: Draft

Trigger Design Pattern

- Purpose of the Class

The SubscriptionTriggerHandler class contains the logic to manage subscription statuses during record creation and updates.

It ensures every subscription has a valid status (Active or Expired) before being saved in Salesforce.

- Business Logic Implemented

Default Assignment:

If Status__c is blank, it is automatically set to 'Active'.

Expiry Check:

If End_Date__c is not null and is earlier than today's date, the subscription is marked as 'Expired'.

File ▾ Edit ▾ Debug ▾ Test ▾ Workspace ▾ Help ▾ < >

SubscriptionHelper.apxc | Log executeAnonymous @9/20/2025, 2:54:07 PM | Log executeAnonymous @9/20/2025, 3:58:15 PM | SubscriptionTriggerHandler.apxc | Log executeAnonymous @9/20/2025, 4:21:42 PM

Code Coverage: None ▾ API Version: 64

```
public class SubscriptionTriggerHandler {

    // Method to handle before insert and before update
    public static void handleBeforeInsertUpdate(List<Subscription_c__c> subs){
        for(Subscription_c__c sub : subs){
            // Set default status if blank
            if(sub.Status__c == null){
                sub.Status__c = 'Active';
            }

            // Check if subscription is expired
            if(sub.End_Date__c != null && sub.End_Date__c < Date.today()){
                sub.Status__c = 'Expired';
            }
        }
    }
}
```

Anonyamous window code

```
Account acc = new Account(Name = 'Test Account for Subscription');
insert acc;

// Create a new subscription (End date in the past)
Subscription_c__c sub1 = new Subscription_c__c(
    Name = 'Expired Subscription',
    Account__c = acc.Id,
    Start_Date__c = Date.today().addDays(-10),
    End_Date__c = Date.today().addDays(-1)
);

// Create a new subscription (End date in the future)
Subscription_c__c sub2 = new Subscription_c__c(
    Name = 'Active Subscription',
     Account__c = acc.Id,               // required
    Start_Date__c = Date.today(),
    End_Date__c = Date.today().addDays(10)
);

// Put them in a list
List<Subscription_c__c> subs = new List<Subscription_c__c>{sub1, sub2};

// Call the handler directly (simulate before insert/update logic)
SubscriptionTriggerHandler.handleBeforeInsertUpdate(subs);

// Insert records into Salesforce
insert subs;

// Query to check the results
List<Subscription_c__c> results = [SELECT Name, Status__c, Start_Date__c,End_Date__c,Account__c FROM Subscription_c__c];
for(Subscription_c__c s : results){
    System.debug('Subscription Name: ' + s.Name + ', Status: ' + s.Status__c);
}
```

Output

File ▾ Edit ▾ Debug ▾ Test ▾ Workspace ▾ Help ▾ < >

SubscriptionHelper.apxc | Log executeAnonymous @9/20/2025, 2:54:07 PM | Log executeAnonymous @9/20/2025, 3:58:15 PM | SubscriptionTriggerHandler.apxc | **Log executeAnonymous @9/20/2025, 4:21:42 PM**

Execution Log

Timestamp	Event	Details
16:21:42:366	USER_DEBUG	[80]|DEBUG|Subscription Name: Test Expired, Status: Draft
16:21:42:366	USER_DEBUG	[80]|DEBUG|Subscription Name: Test Active, Status: Draft
16:21:42:366	USER_DEBUG	[80]|DEBUG|Subscription Name: No End Date, Status: Draft
16:21:42:367	USER_DEBUG	[80]|DEBUG|Subscription Name: Test Expired, Status: Draft
16:21:42:367	USER_DEBUG	[80]|DEBUG|Subscription Name: Test Active, Status: Draft
16:21:42:367	USER_DEBUG	[80]|DEBUG|Subscription Name: No End Date, Status: Draft
16:21:42:367	USER_DEBUG	[80]|DEBUG|Subscription Name: Expired Subscription, Status: Expired
16:21:42:367	USER_DEBUG	[80]|DEBUG|Subscription Name: Active Subscription, Status: Active

Apex Triggers (before/after insert/update/delete)

File ▾ Edit ▾ Debug ▾ Test ▾ Workspace ▾ Help ▾ < >

SubscriptionHelper.apxc | Log executeAnonymous @9/20/2025, 2:54:07 PM | Log executeAnonymous @9/20/2025, 3:58:15 PM | SubscriptionTriggerHandler.apxc | Log executeAnonymous @9/20/2025, 4:21:42 PM | **SubscriptionTrigger.apst *** | Log executeAnonymous @9/20/2025, 4:43:15 PM

Code Coverage: None ▾ API Version: 64 ▾

```
trigger SubscriptionTrigger on Subscription_c__c (before insert, before update) {

    // Call the handler for before insert and before update
    if(Trigger.isBefore){
        if(Trigger.isInsert || Trigger.isUpdate){
            SubscriptionTriggerHandler.handleBeforeInsertUpdate(Trigger.new);
        }
    }
}
```

Annoyamous window code

```
// Step 1: Create an account to link the subscription
Account acc = new Account(Name = 'Test Account');
insert acc;

// Step 2: Create subscriptions (with required fields)
Subscription_c__c sub1 = new Subscription_c__c(
    Name = 'Expired Subscription',
    Account__c = acc.Id,
    Start_Date__c = Date.today().addDays(-10),
    End_Date__c = Date.today().addDays(-1)      // should become 'Expired'
);

Subscription_c__c sub2 = new Subscription_c__c(
    Name = 'Active Subscription',
    Account__c = acc.Id,
    Start_Date__c = Date.today().addDays(-5),
    End_Date__c = Date.today().addDays(5)     // should stay 'Active'
);

// Step 3: Insert subscriptions (trigger will run automatically)
insert new List<Subscription_c__c>{sub1, sub2};

// Step 4: Query to check initial results
List<Subscription_c__c> results = [SELECT Name, Status__c, Start_Date__c, End_Date__c FROM Subscription_c__c];
for(Subscription_c__c s : results){
    System.debug('Before Update - Name: ' + s.Name + ', Status: ' + s.Status__c);
}

// Step 5: Update subscription to simulate status change
sub2.End_Date__c = Date.today().addDays(-1);  // make active subscription expired
update sub2;

// Step 6: Query again to check updated results
results = [SELECT Name, Status__c, Start_Date__c, End_Date__c FROM Subscription_c__c];
for(Subscription_c__c s : results){
    System.debug('After Update - Name: ' + s.Name + ', Status: ' + s.Status__c);
}
```

Output

Execution Log

Timestamp	Event	Details
16:43:15:232	USER_DEBUG	[109]|DEBUG|Before Update - Name: Test Expired, Status: Draft
16:43:15:232	USER_DEBUG	[109]|DEBUG|Before Update - Name: Test Active, Status: Draft
16:43:15:232	USER_DEBUG	[109]|DEBUG|Before Update - Name: No End Date, Status: Draft
16:43:15:232	USER_DEBUG	[109]|DEBUG|Before Update - Name: Test Expired, Status: Draft
16:43:15:232	USER_DEBUG	[109]|DEBUG|Before Update - Name: Test Active, Status: Draft
16:43:15:233	USER_DEBUG	[109]|DEBUG|Before Update - Name: No End Date, Status: Draft
16:43:15:233	USER_DEBUG	[109]|DEBUG|Before Update - Name: Expired Subscription, Status: Draft
16:43:15:233	USER_DEBUG	[109]|DEBUG|Before Update - Name: Active Subscription, Status: Draft
16:43:15:233	USER_DEBUG	[109]|DEBUG|Before Update - Name: Expired Subscription, Status: Expired
16:43:15:233	USER_DEBUG	[109]|DEBUG|Before Update - Name: Active Subscription, Status: Active
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: Test Expired, Status: Draft
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: Test Active, Status: Draft
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: No End Date, Status: Draft
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: Test Expired, Status: Draft
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: Test Active, Status: Draft
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: No End Date, Status: Draft
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: Expired Subscription, Status: Draft
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: Active Subscription, Status: Draft
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: Expired Subscription, Status: Expired
16:43:15:295	USER_DEBUG	[119]|DEBUG|After Update - Name: Active Subscription, Status: Active

SOQL & SOSL -

SOQL.

Purpose of the Code

1. The code retrieves all active subscriptions from Salesforce using a SOQL query.
2. It helps monitor and validate which subscriptions are currently marked as “Active”.
3. Query Details (SOQL)
4. The query fetches key fields:
    a. Name
    b. Status__c
    c. Start_Date__c
    d. End_Date__c
    e. Account__c
5. Filters results where Status__c = 'Active'.

Debugging / Verification

1. After fetching the data, the code loops through each subscription.
2. Uses System.debug() to print subscription details in the debug log.
3. This allows developers/admins to verify the correctness of data and applied business logic.

```
// Query all Active Subscriptions
List<Subscription_c__c> activeSubs = [
    SELECT Name, Status__c, Start_Date__c, End_Date__c, Account__c
    FROM Subscription_c__c
    WHERE Status__c = 'Active'
];

// Debug the results
for(Subscription_c__c sub : activeSubs){
    System.debug(
        'Active Subscription → Name: ' + sub.Name +
        ', Status: ' + sub.Status__c +
        ', Start Date: ' + sub.Start_Date__c +
        ', End Date: ' + sub.End_Date__c +
        ', Account: ' + sub.Account__c
    );
}
```

Execution Log

Timestamp	Event	Details
17:23:29:018	USER_DEBUG	[133]|DEBUG|Active Subscription → Name: Active Subscription, Status: Active, Start Date: 2025-09-20 00:00:00, End Date: 2025-09-30 00:00:00, Account: 001gK00000L4ziNQAR

❖ SOSL –

- **Purpose of the Code**

1. This SOSL (Salesforce Object Search Language) query is used to search records in multiple objects (Subscription__c and Account) at the same time.
2. The search keyword 'Test*' retrieves records where the name starts with "Test".

- **Working of the Code**

1. FIND 'Test*' IN ALL FIELDS → searches across all fields of the objects.
2. RETURNING Subscription__c(Name, Status__c), Account(Name) → specifies which objects and fields to return.
3. The results are stored in a List<List<SObject>>, because multiple objects are being queried.

- **Casting the Results**

1. The results are separated and cast into specific lists:
   - List<Subscription__c> → holds subscription records.
   - List<Account> → holds account records.
2. This ensures type safety and makes it easier to work with the retrieved records.

- **Debugging & Output**

1. System.debug() is used to display the results of both subscriptions and accounts.
2. Example debug output shows the subscription name, status, and the account name for easy verification.

```
List<List<SObject>> searchResults = [
    FIND 'Test*'
    IN ALL FIELDS
    RETURNING
        Subscription_c__c(Name, Status__c),
        Account(Name)
];

List<Subscription_c__c> foundSub = (List<Subscription_c__c>)searchResults[0];
List<Account> foundAccounts = (List<Account>)searchResults[1];

// Extract results
List<Subscription_c__c> foundSubs = (List<Subscription_c__c>)searchResults[0];
List<Account> foundAccount = (List<Account>)searchResults[1];

// Debug Subscriptions
for(Subscription_c__c sub : foundSub){
    System.debug('Found Subscription → Name: ' + sub.Name + ', Status: ' + sub.Status__c);
}

// Debug Accounts
for(Account acc : foundAccounts){
    System.debug('Found Account → Name: ' + acc.Name);
}
```

Anonymous @9/20/2025, 5:23:29 PM | Log executeAnonymous @9/20/2025, 11:28:27 PM | Log

Execution Log

Timestamp	Event	Details
23:29:09:299	USER_DEBUG	[167]|DEBUG|Found Account → Name: Test Account
23:29:09:299	USER_DEBUG	[167]|DEBUG|Found Account → Name: Test Account
23:29:09:299	USER_DEBUG	[167]|DEBUG|Found Account → Name: Test Account for Subscription
23:29:09:299	USER_DEBUG	[167]|DEBUG|Found Account → Name: Test Account for Subscription
23:29:09:299	USER_DEBUG	[167]|DEBUG|Found Account → Name: Test Account for Subscription

Collections: List, Set, Map

List –

- **Storage of Results**

1. **In SOSL, the query can return records from multiple objects at the same time.**
2. **To store these results, Salesforce uses a List<List<SObject>> data structure.**
3. **Each inner List<SObject> represents records from one object (e.g., one list for Subscription__c and another for Account).**

- **Casting to Specific Lists**

1. **The generic SObject results are later cast into specific object lists for ease of use:**
2. **List<Subscription__c> → holds subscription records.**
3. **List<Account> → holds account records.**

Set, Map

1. In this project, I mainly used **Lists** (especially in SOSL and SOQL) for storing and processing records.
2. **Set** and **Map** were **not used** in the current implementation of the Subscription Tracker.
3. A **Set** could be applied in future enhancements to store unique values (e.g., Account Ids) and avoid duplicate processing.
4. A **Map** could be useful to link key-value pairs (e.g., Account Id → Account record) for faster lookups, but it was not required in this project's scope.
5. Even though not implemented here, both collections are important for handling larger datasets and improving efficiency in Salesforce development.

Control statements –

1. In my project, I have used **Control Statements** like for loops and if-else conditions across multiple parts of the code.
2. **For Loops** were used to iterate over records:
3. In **SOQL** and **SOSL**, I used for loops to go through subscription and account records and print their details using System.debug().
4. In **Triggers** and **Classes**, for loops were used to process multiple subscription records in bulk.
5. **If-Else Conditions** were applied to handle decision-making:
6. For example, checking if a subscription's End_Date__c is less than today, then marking it as **Expired**.
7. Also used in classes and triggers to assign default values (e.g., setting status to **Active** if not provided).
8. These control statements helped in making the project **dynamic, flexible, and bulk-safe**.
9. Using for and if-else ensured the code could handle multiple subscriptions at once while applying the right business logic.

Asynchronous Processing

- **Batch apex**

1. The **SubscriptionExpiryBatch** automates status updates for expired subscriptions in the tracker.
2. Implements **Batch Apex with Stateful** to process large volumes and maintain counts (processed, updated, failed).
3. Uses **SOQL with runtime binding** to fetch active subscriptions past their end date.
4. Ensures **bulk-safe updates** with partial success handling and detailed error logging.
   Provides a **final summary** of execution for monitoring subscription data accuracy.
5. Enhances the **subscription tracker's reliability** by keeping statuses up to date automatically.

```apex
public class SubscriptionExpiryBatch implements Database.Batchable<SObject>, Database.Stateful {
    // stateful counters (persist across execute() calls)
    public Integer totalProcessed = 0;
    public Integer totalUpdated = 0;
    public Integer totalFailed = 0;

    // START: build the QueryLocator - returns subscriptions that need expiring
    public Database.QueryLocator start(Database.BatchableContext bc) {
        // Bind Date.today() so the query is evaluated at runtime
        return Database.getQueryLocator([
            SELECT Id, Name, Status__c, Start_Date__c, End_Date__c, Account__c
            FROM Subscription_c__c
            WHERE End_Date__c < :Date.today()
              AND (Status__c = null OR Status__c != 'Expired')
        ]);
    }

    // EXECUTE: called for each batch "scope"
    public void execute(Database.BatchableContext bc, List<SObject> scope) {
        List<Subscription_c__c> scopeSubs = (List<Subscription_c__c>) scope;
        List<Subscription_c__c> toUpdate = new List<Subscription_c__c>();

        // Prepare updates in memory
        for (Subscription_c__c s : scopeSubs) {
            totalProcessed++;
            // Defensive check (query already filtered but good to double-check)
            if (s.End_Date__c != null && s.End_Date__c < Date.today()) {
                s.Status__c = 'Expired';
                toUpdate.add(s);
            }
        }

        // Bulk update with partial success allowed
        if (!toUpdate.isEmpty()) {
            Database.SaveResult[] results = Database.update(toUpdate, false);
            for (Integer i = 0; i < results.size(); i++) {
                if (results[i].isSuccess()) {
                    totalUpdated++;
                } else {
                    totalFailed++;
                    // Log each error for debugging
                    for (Database.Error err : results[i].getErrors()) {
                        System.debug(
                            'Failed to update Subscription Id=' + toUpdate[i].Id +
                            ' : ' + err.getStatusCode() + ' - ' + err.getMessage()
                        );
                    }
                }
            }
        }
    }

    // FINISH: run after all batches finish
    public void finish(Database.BatchableContext bc) {
        // Summary log — replace with email/PlatformEvent/Queueable if you want notifications
        System.debug('SubscriptionExpiryBatch completed. Processed: ' + totalProcessed +
                     ', Updated: ' + totalUpdated + ', Failed: ' + totalFailed);
    }
}
```

```apex
// Run the batch with batch size = 200 (you can change this number)
Id jobId = Database.executeBatch(new SubscriptionExpiryBatch(), 100);

// Debug log with the Job Id so you can track it
System.debug('Batch job started. Job Id = ' + jobId);
```

OUTPUT -

Execution Log

Timestamp	Event	Details
17:23:29:018	USER_DEBUG	[133]|DEBUG|Active Subscription → Name: Active Subscription, Status: Active, Start Date: 2025-09-20 00:00:00, End Date: 2025-09-30 00:00:00, Account: 001gK00000L4ziNQAR

- **<u>Queue apex</u>**

1. **UpdateExpiredSubscriptions** class automates marking expired subscriptions as "Expired".
2. Uses **Queueable Apex** for asynchronous execution and better performance than triggers for bulk updates.
3. Fetches all subscriptions with **End_Date__c <= today** that are not already expired.
4. Ensures data accuracy by **updating statuses in bulk**.
5. Can be **chained or scheduled** for regular execution, making the subscription tracker more reliable.

```apex
public class UpdateExpiredSubscriptions implements Queueable {

    public void execute(QueueableContext context) {
        // Query all subscriptions that have expired but are not yet marked as Expired
        List<Subscription_c__c> expiredSubs = [
            SELECT Id, Status__c, End_Date__c
            FROM Subscription_c__c
            WHERE Status__c != 'Expired' AND End_Date__c <= :Date.today()
        ];

        // Update status
        for(Subscription_c__c sub : expiredSubs) {
            sub.Status__c = 'Expired';
        }

        if(!expiredSubs.isEmpty()) {
            update expiredSubs;
        }
    }
}
```

Execution Log

Timestamp	Event	Details
01:29:51:049	USER_DEBUG	[181]|DEBUG|Queueable Job ID: 707gK00000DlDow

Scheduled Apex

- Not required because the batch/queueable jobs can be run **on-demand** or scheduled externally if needed.
- Project scope was limited, so continuous daily/weekly scheduling was not implemented.

Future Methods

- Queueable Apex was chosen instead of Future methods since it is **more flexible**, supports chaining, and is better for bulk updates.
- Future methods are limited and not suitable for updating large sets of records in this project.

Exception Handling

- The project focused mainly on **core functionality** (status updates) and not on advanced error recovery.
- Basic error logs using System.debug were sufficient for the prototype stage.

Test Classes

- As this was a **project-level implementation**, formal unit test classes were not added.
- In a real production scenario, test classes would be mandatory to ensure **code coverage** and **reliability**.
