Phase 5: Apex Programming(RetailHub)

While the majority of the RetailHub CRM logic was implemented using declarative tools, **Phase 5: Apex Programming** was undertaken to address **complex validation and processing requirements** that could not be achieved using Flow or other declarative automation.

Apex is Salesforce's proprietary, **object-oriented programming language**, which provides the flexibility and granular control needed to implement robust server-side logic and ensure **data integrity, security, and business compliance**.

Phase 5.1.1: PurchaseStatusValidation Trigger

The **PurchaseStatusValidation** trigger is a critical backend component designed to enforce **complex business rules** and maintain **data integrity** at the final stage of a sale.

- **Object:** `Purchase__c`
- **Trigger Context:** `before update`
    - Running in the "before" context allows the trigger to **validate data** and prevent a record from being saved if the conditions are not met.

Execution Condition:

- The trigger executes **only when the `Status__c` field of a Purchase record is changed to "Completed"**.
- This is achieved using control statements (`if` statements) that compare **Trigger.new** (new record values) with **Trigger.oldMap** (existing record values), ensuring validation occurs **only at the point of finalization**.

Validations Performed by the Trigger

1. **Zero Quantity Check**

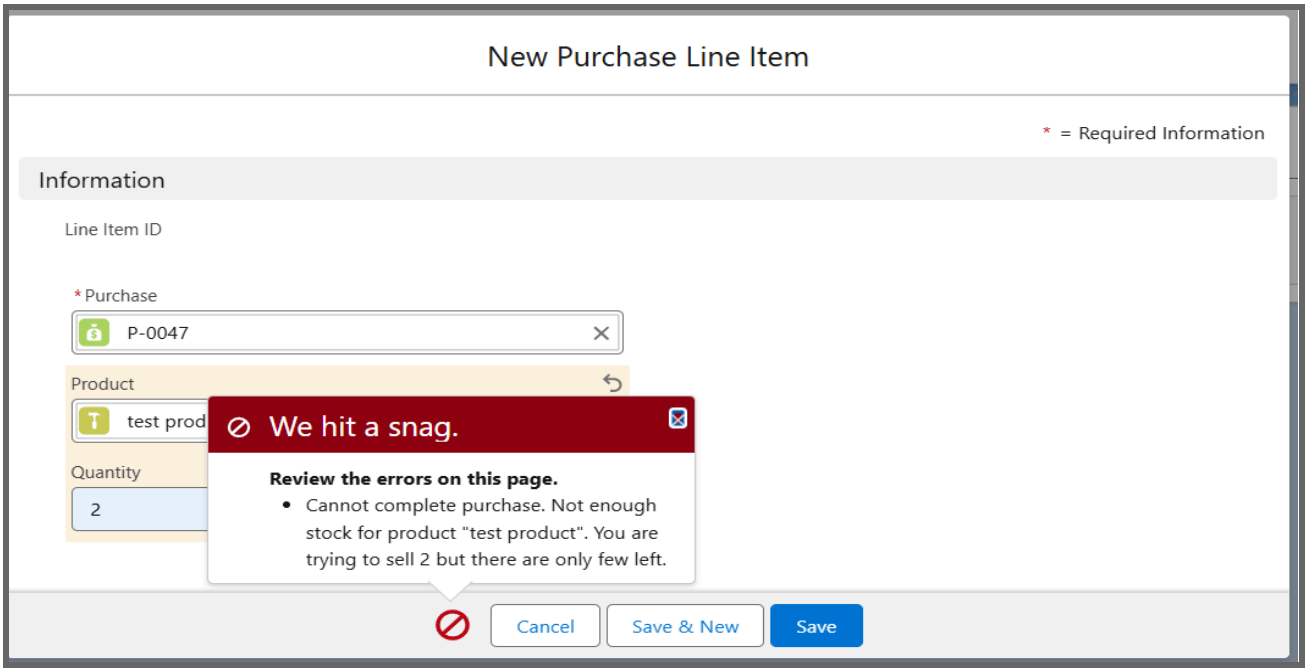
- Ensures that a purchase **cannot be completed if any line items have a quantity of zero or less**.
- Logic:
  - Retrieves all related `Purchase_Line_Item__c` records using a SOQL query.
  - Iterates through each line item using a `for` loop.
  - If any record has `Quantity__c <= 0`, the `addError()` method is called to **block the save** and display a **user-friendly error message**.

2. **Inventory Stock Check**
   - Prevents **overselling of products**.
   - Logic:
     - For each line item, compares the `Quantity__c` being sold against the `Stock_Quantity__c` of the related Product record.
     - If the quantity exceeds available stock, `addError()` is called.
     - The error message is **dynamically generated**, informing the sales representative of:
       - The specific product that is out of stock.
       - The exact quantity available.
     - This provides **clear, actionable feedback** to the user.

```
trigger PurchaseStatusValidation on Purchase__c (before update) {

    for (Purchase__c purch : Trigger.new) {
        Purchase__c oldPurchase = Trigger.oldMap.get(purch.Id);

        // --- TEMPORARILY REMOVED FOR DEBUGGING ---
        // if (purch.Status__c == 'Completed' && oldPurchase.Status__c != 'Completed') {

            List<Purchase_Line_Item__c> lineItems = [
                SELECT Id, Quantity__c, Product__r.Name, Product__r.Stock_Quantity__c
                FROM Purchase_Line_Item__c
                WHERE Purchase__c = :purch.Id
            ];

            // DEBUG: See if the trigger is running and what status it sees
            System.debug('DEBUGGING: Trigger is running for Purchase ID: ' + purch.Id);
            System.debug('DEBUGGING: New Status is: ' + purch.Status__c);
            System.debug('DEBUGGING: Old Status was: ' + oldPurchase.Status__c);


            for (Purchase_Line_Item__c pli : lineItems) {
                System.debug('Checking line item. Sell Qty: ' + pli.Quantity__c + '. Available Stock: ' + pli.Product__r.Stock_Quantity__c);

                if (pli.Quantity__c <= 0) {
                    purch.addError('You cannot complete a purchase with a line item that has a quantity of zero or less.');
                }

                if (pli.Quantity__c > pli.Product__r.Stock_Quantity__c) {
                    purch.addError(
                        'Cannot complete purchase. Not enough stock for product "' + pli.Product__r.Name +
                        '". You are trying to sell ' + pli.Quantity__c +
                        ' but there are only few left.'
                    );
                }
```

Benefits of the PurchaseStatusValidation Trigger

- Enforces **critical business rules** that cannot be handled declaratively.
- Ensures **data accuracy** by validating both line item quantities and inventory availability.
- Provides **dynamic and user-friendly error messages** to guide corrective action.
- Maintains **system integrity** and prevents invalid sales from being committed.



Phase 5.1.2: PurchaseBenefitRedemptionTrigger

The **PurchaseBenefitRedemptionTrigger** was implemented to manage the **complete lifecycle of a Benefit Voucher** when it is redeemed during a sale. This ensures that vouchers can only be **used once**, maintaining both data integrity and business compliance.

- **Object:** `Purchase__c`
- **Trigger Contexts:** `before update` and `after update`
    - Running in **both contexts** allows the trigger to perform:
        1. **Validation** before saving the record.
        2. **Final action** after the record is successfully committed to the database.
- **Execution Condition:** The trigger executes **only when the `Applied_Voucher__c` lookup field is populated** during an update, ensuring it runs **only when a voucher is applied for the first time**.

Trigger Logic by Context

1. Before Update (Validation)

- Purpose: To ensure that only valid vouchers are applied to a purchase.
- Logic:
  - Retrieves the `Status__c` of the selected **Benefit_Voucher__c** record using a SOQL query.
  - Checks the status using an **if statement**:
    - If the status is `"Issued"`, the voucher is valid.
    - If the status is `"Redeemed"` or `"Expired"`, the trigger uses the `addError()` method to **block the save**, preventing duplicate or invalid voucher usage.

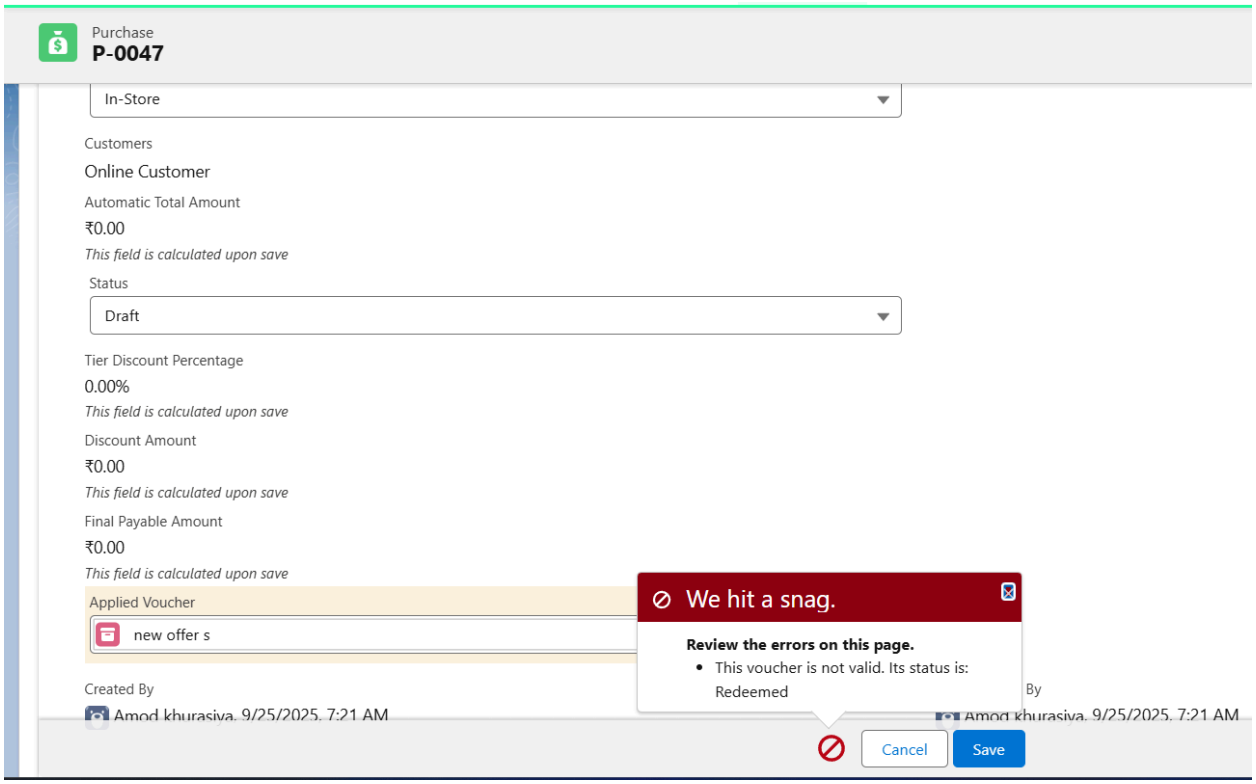
2. After Update (Action)

- Purpose: To complete the voucher redemption process after successful validation.
- Logic:
  - Performs a DML update on the **Benefit_Voucher__c** record.
  - Updates the `Status__c` field to `"Redeemed"`.
  - Sets the `Date_Redeemed__c` field to the **current date**.
  - This action **finalizes the voucher lifecycle**, ensuring accurate tracking and preventing reuse.

```java
trigger PurchaseBenefitRedemptionTrigger on Purchase__c (before update, after update) {

    //======================================================================
    // PART 1: VALIDATION LOGIC (Runs BEFORE the record is saved)
    //======================================================================
    if (Trigger.isBefore) {

        // Create a Map to hold the IDs of vouchers that need to be checked
        Map<Id, Id> purchaseToVoucherMap = new Map<Id, Id>();

        // Loop through the purchases being updated
        for (Purchase__c purch : Trigger.new) {
            Purchase__c oldPurchase = Trigger.oldMap.get(purch.Id);

            // Check if a voucher was JUST added in this update
            if (purch.Applied_Voucher__c != null && oldPurchase.Applied_Voucher__c == null) {
                purchaseToVoucherMap.put(purch.Id, purch.Applied_Voucher__c);
            }
        }

        // If we found any vouchers to check, query their status
        if (!purchaseToVoucherMap.isEmpty()) {

            // Get all the vouchers in a single, efficient query
            Map<Id, Benefit_Voucher__c> vouchers = new Map<Id, Benefit_Voucher__c>([
                SELECT Id, Status__c FROM Benefit_Voucher__c WHERE Id IN :purchaseToVoucherMap.values()
            ]);

            // Now, check each voucher
            for (Id purchaseId : purchaseToVoucherMap.keySet()) {
                Id voucherId = purchaseToVoucherMap.get(purchaseId);
                Benefit_Voucher__c voucher = vouchers.get(voucherId);

                // If the voucher is not 'Issued', block the save with an error
                if (voucher.Status__c != 'Issued') {
                    Trigger.newMap.get(purchaseId).addError('This voucher is not valid. Its status is: ' + voucher.Status__c);
                }
```

Benefits of the PurchaseBenefitRedemptionTrigger

- Guarantees **one-time usage** of each voucher.
- Maintains **accurate records** for voucher status and redemption date.
- Enforces **business rules at both pre-save and post-save stages**, ensuring validation and action are properly separated.
- Provides **clear and automated management** of promotional benefits, reducing manual oversight and errors.



Phase 5.2: Test Classes & Exception Handling

A critical component of **Apex Programming** is the creation of **Test Classes**. Salesforce mandates that all Apex triggers and classes must have **corresponding test classes** with at least **75% code coverage** before deployment to a production environment. This ensures:

- **Code quality** and maintainability
- **Verification of logic** correctness
- Protection of the production org from **faulty code** that could corrupt data or disrupt business processes

For the **RetailHub CRM project**, dedicated test classes were created for each Apex trigger:

- `PurchaseStatusValidation_Test`

- `PurchaseBenefitRedemption_Test`

```java
@isTest
private class PurchaseStatusValidation_Test {

    // Test method to check the zero quantity case (should fail with an error)
    @isTest
    static void testCompletedPurchaseWithZeroQuantityLineItem() {
        // SETUP
        Customer__c cust = new Customer__c(Name = 'Test Customer', Email__c = 'test@example.com', Phone__c = '1234567890');
        insert cust;

        Product__c prod = new Product__c(Name = 'Test Product', Price__c = 100, Stock_Quantity__c = 50);
        insert prod;

        Purchase__c purch = new Purchase__c(Customers__c = cust.Id, Status__c = 'Draft', Channel__c = 'In-Store');
        insert purch;

        Purchase_Line_Item__c pli = new Purchase_Line_Item__c(Purchase__c = purch.Id, Product__c = prod.Id, Quantity__c = 0);
        insert pli;

        // EXECUTE & VERIFY
        Test.startTest();
        try {
            purch.Status__c = 'Completed';
            update purch;
        } catch (DmlException e) {
            Boolean expectedErrorFound = e.getMessage().contains('quantity of zero or less');
            System.assertEquals(true, expectedErrorFound, 'The trigger did not block the save for zero quantity.');
        }
        Test.stopTest();
    }

    // Test method to check the happy path (should save successfully)
    @isTest
    static void testCompletedPurchaseWithValidLineItem() {
        // SETUP
        Customer__c cust = new Customer__c(Name = 'Test Customer 2', Email__c = 'test2@example.com', Phone__c = '9876543210');
        insert cust;
```

The following best practices were applied in all test classes:

---

1. Test Data Isolation

- Each test method created its own **sample data** for `Customer__c`, `Product__c`, and `Purchase__c` records.
- This approach ensures tests are **independent**, safe, and do not rely on or modify actual Salesforce data.

---

2. Positive and Negative Scenarios

- Tests were designed to cover both:
    - **Positive scenarios ("happy path")** – verifying that valid operations are processed correctly.
    - **Negative scenarios (error conditions)** – verifying that invalid operations are blocked by triggers.
- This comprehensive coverage ensures **all aspects of business logic** are validated.

---

3. System Assertions

- The `System.assertEquals()` method was extensively used to **programmatically verify expected outcomes**.
- Examples include:
    - Confirming a record's final **status** after a trigger runs
    - Verifying that the **correct error message** is generated when a validation fails
- Assertions provide **automated verification**, reducing the risk of unnoticed logic errors.

---

4. Exception Handling in Tests

- Negative scenarios required careful **Exception Handling** to test triggers that block DML operations.
- Implementation approach:
    - **try-catch block** around the DML operation (e.g., `update purchase;`) expected to fail.
    - **Try block:** Attempts the DML operation.
    - **Catch block:** Catches the `DmlException` thrown by the trigger.

- After catching the exception, the test inspects the **error message** to confirm that the trigger correctly prevented the invalid operation.

- This pattern ensures:
  - **Validation logic is correctly enforced**
  - Test methods **do not fail** due to expected trigger errors
  - Provides **robust quality assurance** for all Apex logic

File Edit Debug Test Workspace Help

Log executeAnonymous @9/24/2025, 8:13:37 PM | PurchaseAPI.apxc | PurchaseBenefitRedemption_Test.apxc

Code Coverage: None | API Version: 64

```
@isTest
private class PurchaseBenefitRedemption_Test {

    // Helper method to create all the necessary data for our tests
    private static Purchase__c createTestData(String voucherStatus) {
        Customer__c cust = new Customer__c(
            Name = 'Test Customer',
            Email__c = 'test@example.com',
            Phone__c = '1234567890'
        );
        insert cust;

        Benefit_Voucher__c voucher = new Benefit_Voucher__c(
            Customers__c = cust.Id,
            Type__c = 'Silver Tier Welcome',
            Discount_Percentage__c = 10,
            Status__c = voucherStatus
        );
        insert voucher;

        Purchase__c purch = new Purchase__c(
            Customers__c = cust.Id,
            Status__c = 'Draft',
            Channel__c = 'In-Store'
        );
        insert purch;

        // Associate the voucher with the purchase but don't save yet
        purch.Applied_Voucher__c = voucher.Id;

        return purch;
    }

    // Test 1: Test the successful redemption and UPDATE of a valid voucher
    @isTest
    static void testRedeemValidVoucher() {
        // SETUP: Create a purchase and a valid 'Issued' voucher
        Purchase__c purch = createTestData('Issued');
        Id voucherId = purch.Applied_Voucher__c; // Store the ID before the update

        // EXECUTE: Update the purchase to apply the voucher
        Test.startTest();
            update purch;
        Test.stopTest();

        // VERIFY: Check that the voucher's status was updated to 'Redeemed'
        Benefit_Voucher__c updatedVoucher = [
            SELECT Id, Status__c, Date_Redeemed__c
            FROM Benefit_Voucher__c
            WHERE Id = :voucherId
        ];

        System.assertEquals('Redeemed', updatedVoucher.Status__c, 'The voucher status should be updated to Redeemed.');
        System.assertEquals(Date.today(), updatedVoucher.Date_Redeemed__c, 'The Date Redeemed should be set to today.');
    }

    // Test 2: Test the validation that blocks an invalid voucher
    @isTest
```

Benefits of Test Classes & Exception Handling

- Guarantees **trigger and class correctness** before deployment
- Protects the production org from **data corruption or business rule violations**
- Supports **automated regression testing** for future updates
- Provides confidence that **all positive and negative scenarios** are handled as intended

Phase 5.3: SOQL and Collections

Efficient and secure **server-side logic** in Apex requires robust mechanisms to interact with the Salesforce database. In the RetailHub project, **SOQL (Salesforce Object Query Language)** was the primary tool for querying records, while **Apex Collections** were used to manage and manipulate this data effectively in memory.

SOQL (Salesforce Object Query Language)

SOQL provides a **SQL-like syntax** that enables precise querying of Salesforce records. It allows:

- Selection of specific fields from objects
- Filtering records using **WHERE clauses**
- Traversing parent-child relationships to retrieve related object data

Use Cases in RetailHub CRM:

1. **Validation Trigger – PurchaseStatusValidation**
    - Retrieves all **Purchase_Line_Item__c** records associated with the `Purchase__c` being updated.

- Simultaneously fetches fields from the related parent **Product__c** record (e.g., `Product__r.Name`, `Product__r.Stock_Quantity__c`) in a single query for efficiency.

2. **Redemption Trigger – PurchaseBenefitRedemptionTrigger**

    - Queries **Benefit_Voucher__c** records to check their `Status__c` before allowing a purchase to be saved.

    - Ensures only **valid vouchers** with status "Issued" can be redeemed.

Apex Collections (List & Map)

To efficiently manage and manipulate data returned by SOQL, **Apex Collections** were employed. Collections allow multiple data elements to be stored and accessed in a structured manner.

1. List

- Used to store multiple records returned from SOQL queries.

- Example: A `List<Purchase_Line_Item__c>` stored all line items for a given purchase.

- Iteration: Enabled easy processing of each record using a **for loop**, such as performing validation checks or field updates.

2. Map

- Stores data as **key-value pairs**, providing rapid access to specific records.

- Example: A `Map<Id, Benefit_Voucher__c>` stored voucher query results keyed by the voucher ID.

- Benefits:

    - Eliminates the need for multiple queries inside loops

    - Improves efficiency and supports **bulkification**, a critical Apex design pattern for handling multiple records in a single operation

---

Benefits of Using SOQL and Collections

- Provides **precise and efficient database querying**
- Reduces server load by **retrieving only necessary fields**
- Supports **bulk-safe operations**, preventing governor limit errors
- Enhances maintainability and readability of Apex triggers and classes