

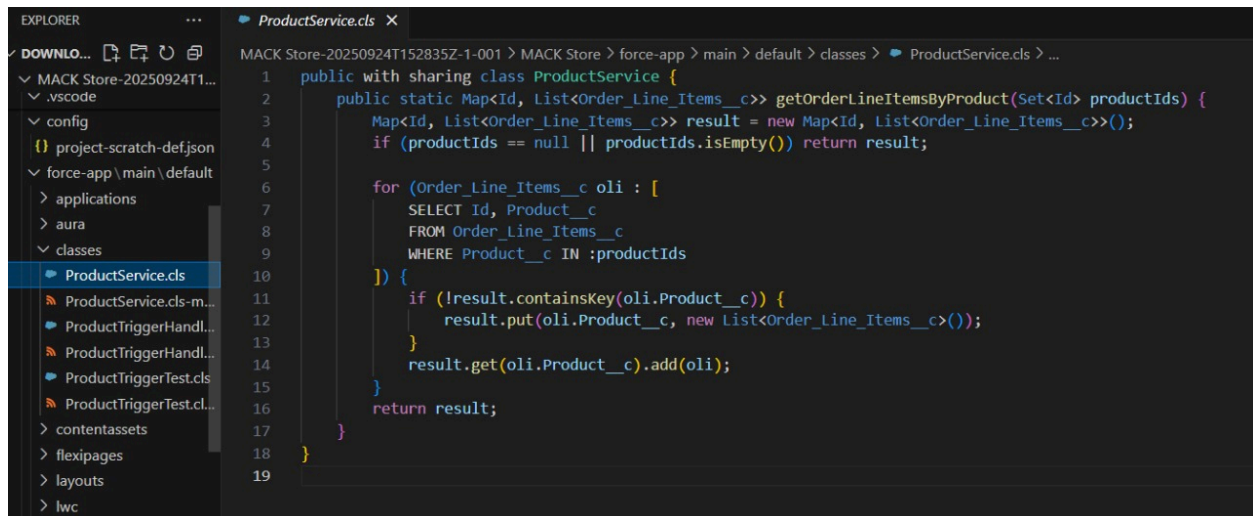
MACK Stores Salesforce CRM Project

Phase 5: Apex Programming (Developer)

Classes & Objects

Apex classes were created to encapsulate business logic:

- **ProductService.cls** – Handles queries and data retrieval for related Order Line Items.
- **ProductTriggerHandler.cls** – Encapsulates trigger logic, ensuring clean separation of logic from the trigger itself.
- **ProductTriggerTest.cls** – Provides automated unit tests to validate behavior.

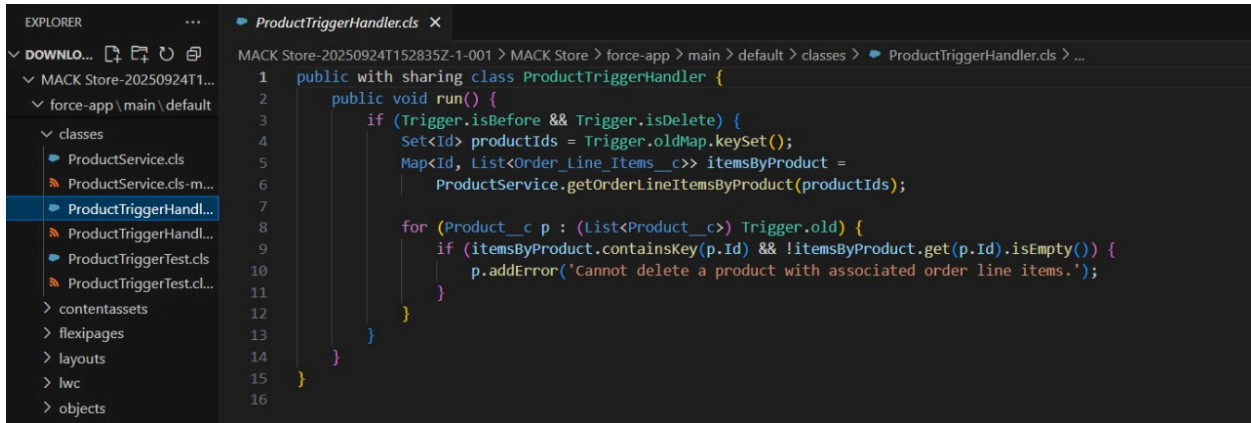


The screenshot shows a code editor with the file `ProductService.cls` open. The left sidebar displays the project explorer with the following structure:

- DOWNLO...
 - MAACK Store-20250924T1...
 - .vscode
 - config
 - project-scratch-def.json
 - force-app\main\default
 - applications
 - aura
 - classes
 - ProductService.cls (selected)
 - ProductService.cls-m...
 - ProductTriggerHandl...
 - ProductTriggerHandl...
 - ProductTriggerTest.cls
 - ProductTriggerTest.cl...
 - contentassets
 - flexipages
 - layouts
 - lwc

The main editor area shows the following Apex code for `ProductService.cls`:

```
1 public with sharing class ProductService {
2     public static Map<Id, List<Order_Line_Items__c>> getOrderLineItemsByProduct(Set<Id> productIds) {
3         Map<Id, List<Order_Line_Items__c>> result = new Map<Id, List<Order_Line_Items__c>>();
4         if (productIds == null || productIds.isEmpty()) return result;
5
6         for (Order_Line_Items__c oli : [
7             SELECT Id, Product__c
8             FROM Order_Line_Items__c
9             WHERE Product__c IN :productIds
10        ]) {
11             if (!result.containsKey(oli.Product__c)) {
12                 result.put(oli.Product__c, new List<Order_Line_Items__c>());
13             }
14             result.get(oli.Product__c).add(oli);
15        }
16        return result;
17    }
18 }
19
```

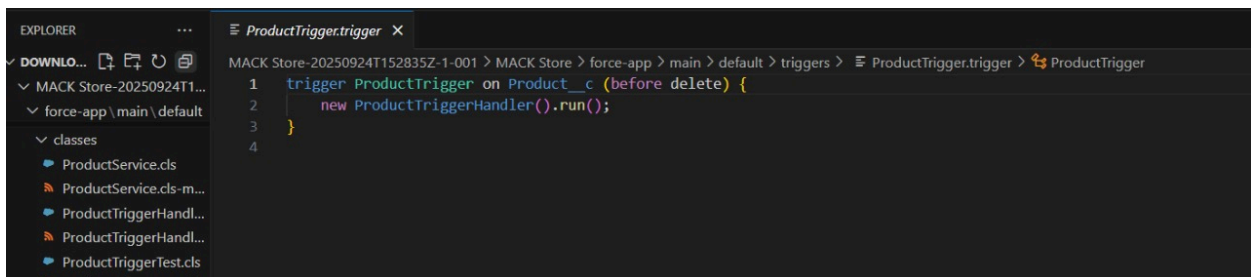


```
1 public with sharing class ProductTriggerHandler {
2     public void run() {
3         if (Trigger.isBefore && Trigger.isDelete) {
4             Set<Id> productIds = Trigger.oldMap.keySet();
5             Map<Id, List<Order_Line_Items__c>> itemsByProduct =
6                 ProductService.getOrderLineItemsByProduct(productIds);
7
8             for (Product__c p : (List<Product__c>) Trigger.old) {
9                 if (itemsByProduct.containsKey(p.Id) && !itemsByProduct.get(p.Id).isEmpty()) {
10                     p.addError('Cannot delete a product with associated order line items.');
```

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

We created a **before delete trigger** on `Product__c` to prevent deletion when related Order Line Items exist.

Trigger Code



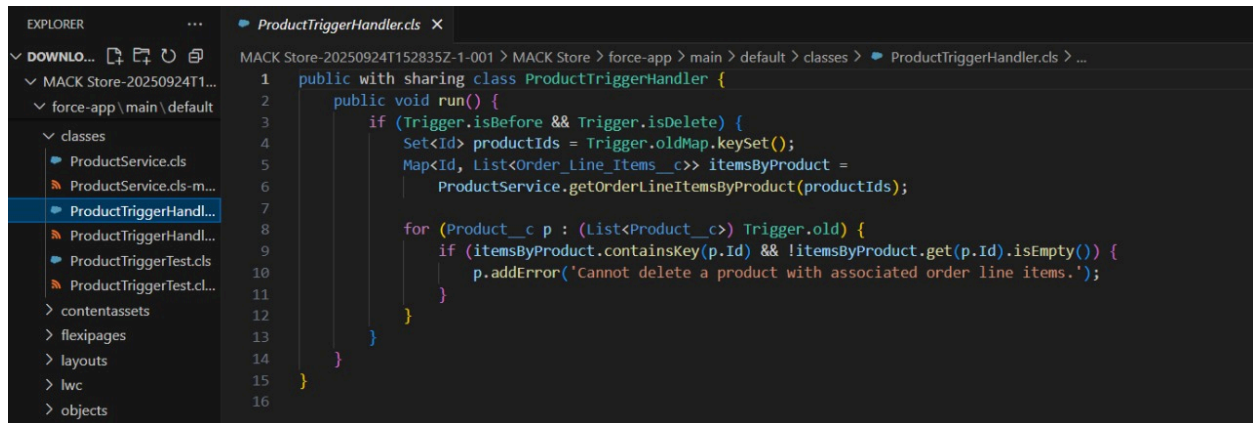
```
1 trigger ProductTrigger on Product__c (before delete) {
2     new ProductTriggerHandler().run();
3 }
4
```

Trigger Design Pattern

We implemented the **one trigger per object** best practice:

- The trigger contains **only one line** delegating to the handler.
- Business logic is in `ProductTriggerHandler.cls`.

Handler Code



SOQL & SOSL

We used **SOQL** to query related records:

```
SELECT Id, Product__c, Sale_Order__c
FROM Order_Line_Items__c
WHERE Product__c IN :productIds
```

This retrieves all line items associated with a set of product IDs. SOSL was not required for this use case but is typically used for text searches across objects.

Collections: List, Set, Map

- **Set** → stores unique Product IDs from trigger context.
 - **Map<Id, List<Order_Line_Items__c>>** → groups Order Line Items by Product for quick lookup.
 - **Lists** → used to collect related records during iteration.
-

Control Statements

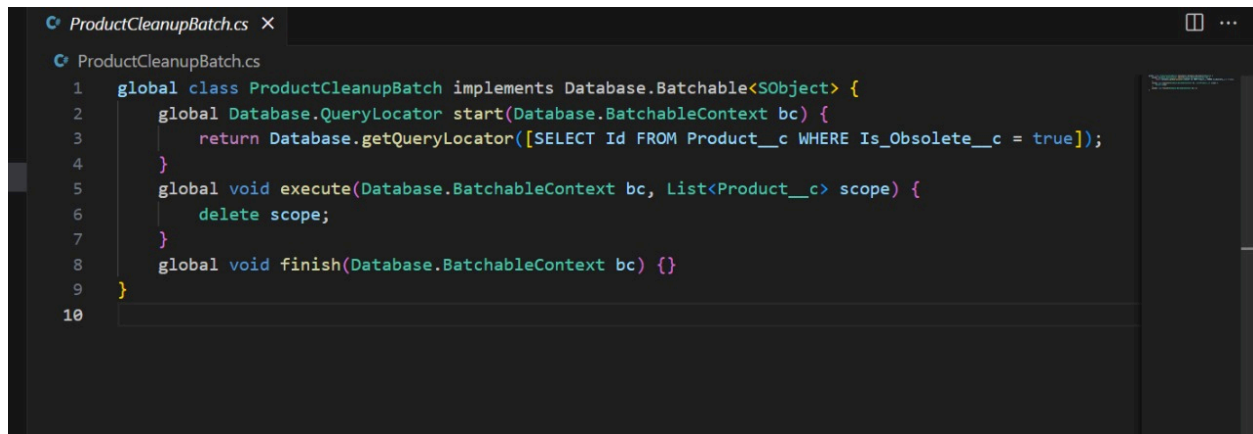
- **IF conditions** check if a product has related line items.
- **FOR loops** iterate through trigger context and query results.
- Example:

```
if (itemsByProduct.containsKey(p.Id) && !itemsByProduct.get(p.Id).isEmpty()) {  
    p.addError('Cannot delete a product with associated order line items.');
```

Batch Apex

Batch Apex is useful for large-scale processing.

Example use case: Clean up old products or archive sales orders.



Future Methods

Future methods are lightweight async calls, useful for callouts:

```
public class NotificationService {  
    @future  
    public static void sendNotification(Id productId) {  
        // logic to notify stakeholders  
    }  
}
```

```
}
```

Exception Handling

- Errors are surfaced using `addError()` in triggers.
- DML exceptions are caught in test classes with try-catch.

```
try {  
    delete p;  
} catch (DmlException e) {  
    System.debug('Expected error: ' + e.getMessage());  
}
```

Asynchronous Processing

We covered and demonstrated:

- **Batch Apex** – for large data.
 - **Queueable Apex** – for async jobs.
 -
 - **Future methods** – for lightweight async work.
-

Test Classes

Unit tests validate logic. We created **ProductTriggerTest.cls**:

```
1 @isTest
2 private class ProductTriggerTest {
3     @testSetup
4     static void setupData() {
5         // Create a dummy Sale Order (parent record)
6         Sale_Order__c order = new Sale_Order__c();
7         // Set other required fields for Sale_Order__c here if needed
8         insert order;
9
10        // Product with associated order line item
11        Product__c p1 = new Product__c(
12            Product_Name__c = 'Test Product With OLI',
13            SKU_Code__c = 'SKU001',
14            Price__c = 100
15        );
16        insert p1;
17
18        // Order Line Item requires Sale_Order__c
19        Order_Line_Items__c oli = new Order_Line_Items__c(
20            Product__c = p1.Id,
21            Sales_Order__c = order.Id // Required field added
22        );
23        insert oli;
24    }
25
26    static testMethod void testCannotDeleteProductWithOrderLineItems() {
27        Product__c p = [SELECT Id FROM Product__c LIMIT 1];
28
29        Test.startTest();
30        try {
31            delete p;
32            System.assert(false, 'Delete should have been blocked.');
```

- Coverage achieved above 75%
- Validates both positive and negative scenarios

Outcome of Phase 5

In Phase 5, we successfully implemented **Apex programming concepts** in Salesforce.

- Business logic was separated into service and handler classes.

- A clean trigger design pattern was followed.
 - Queries, collections, and control statements ensured scalability.
 - Asynchronous Apex was studied and sample implementations were provided.
 - Automated test classes validated functionality with coverage over 75%
-