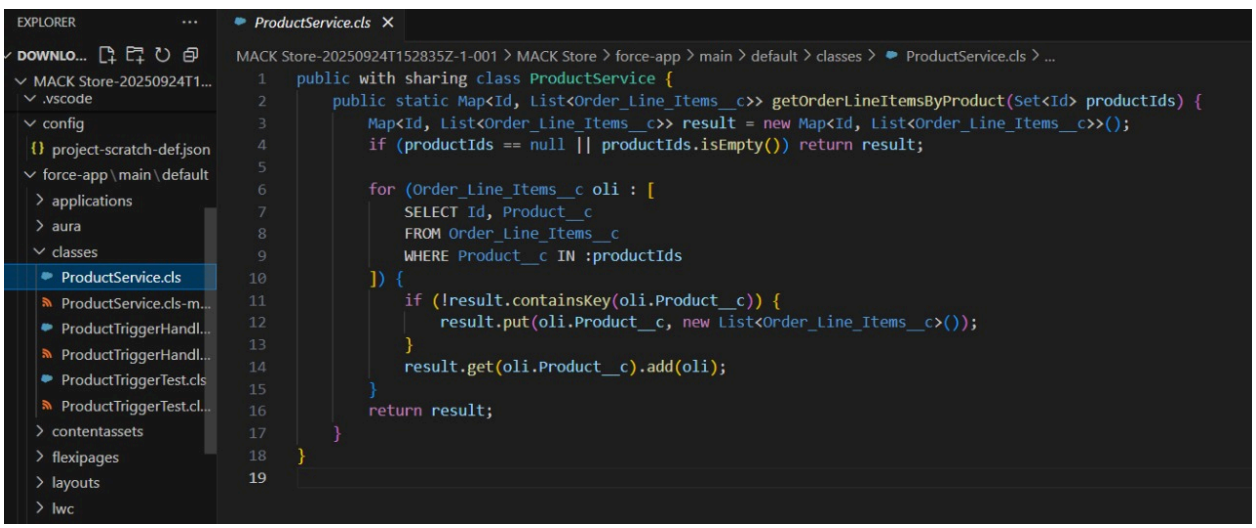
MACK Stores Salesforce CRM Project

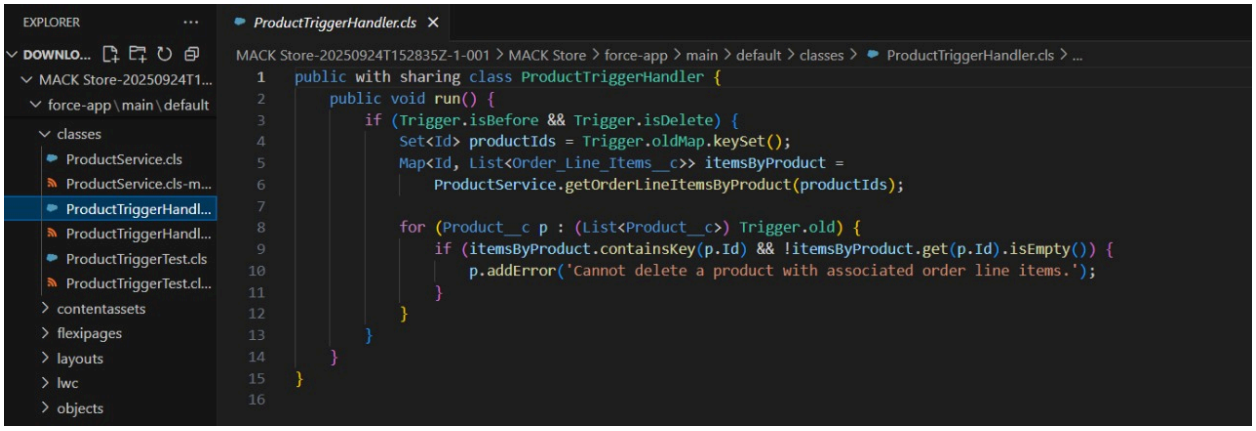
Phase 5: Apex Programming (Developer)

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.

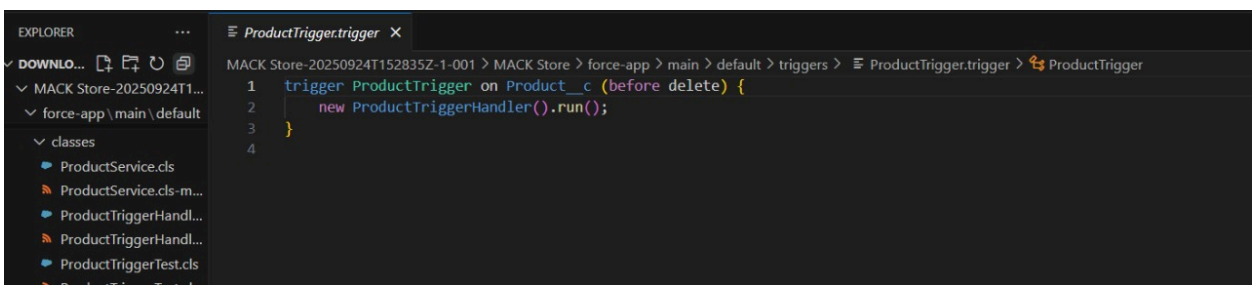




Apex Triggers (before/after insert/update/delete)

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

Trigger Code

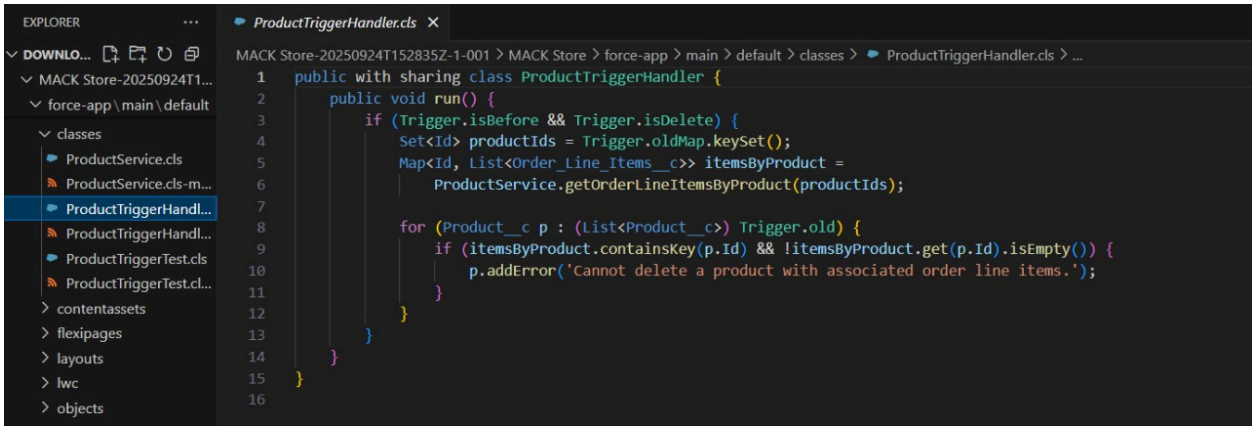


Trigger Design Pattern

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.

ProductCleanupBatch.cs

```
global class ProductCleanupBatch implements Database.Batchable<SObject> {
    global Database.QueryLocator start(Database.BatchableContext bc) {
        return Database.getQueryLocator([SELECT Id FROM Product__c WHERE Is_Obsolete__c = true]);
    }
    global void execute(Database.BatchableContext bc, List<Product__c> scope) {
        delete scope;
    }
    global void finish(Database.BatchableContext bc) {}
}
```

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());
}
```

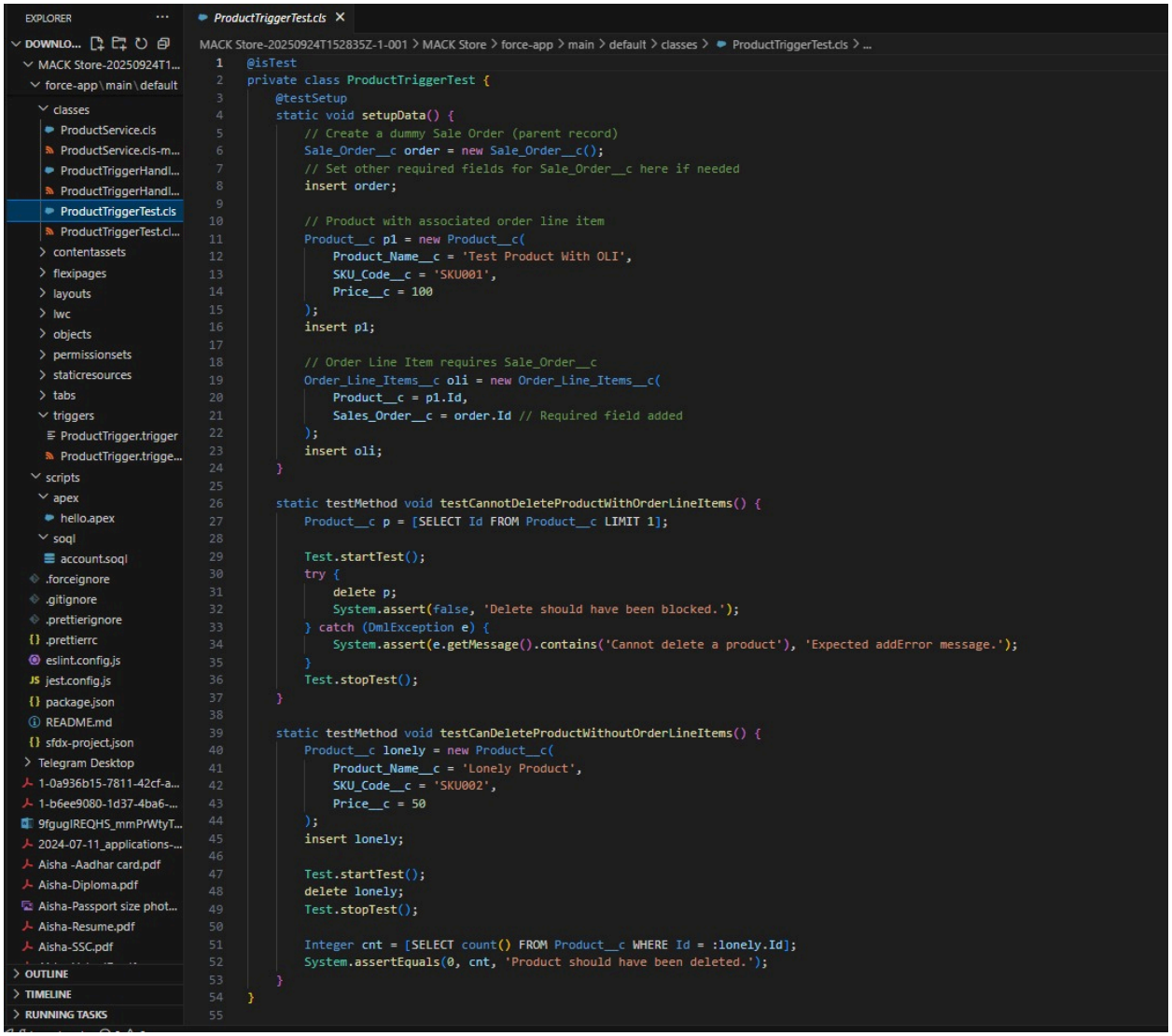
Asynchronous Processing

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**:



- **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%
