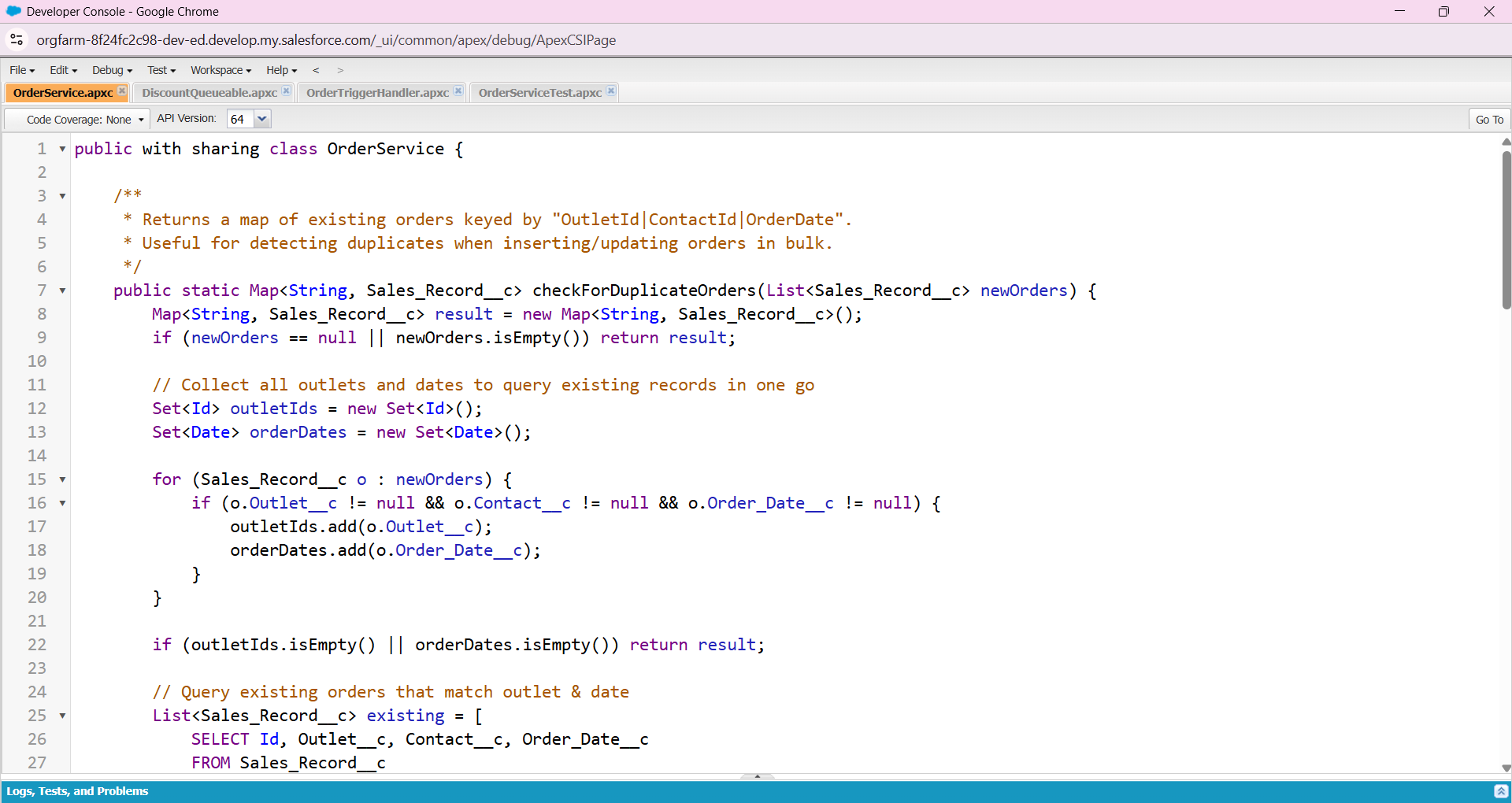
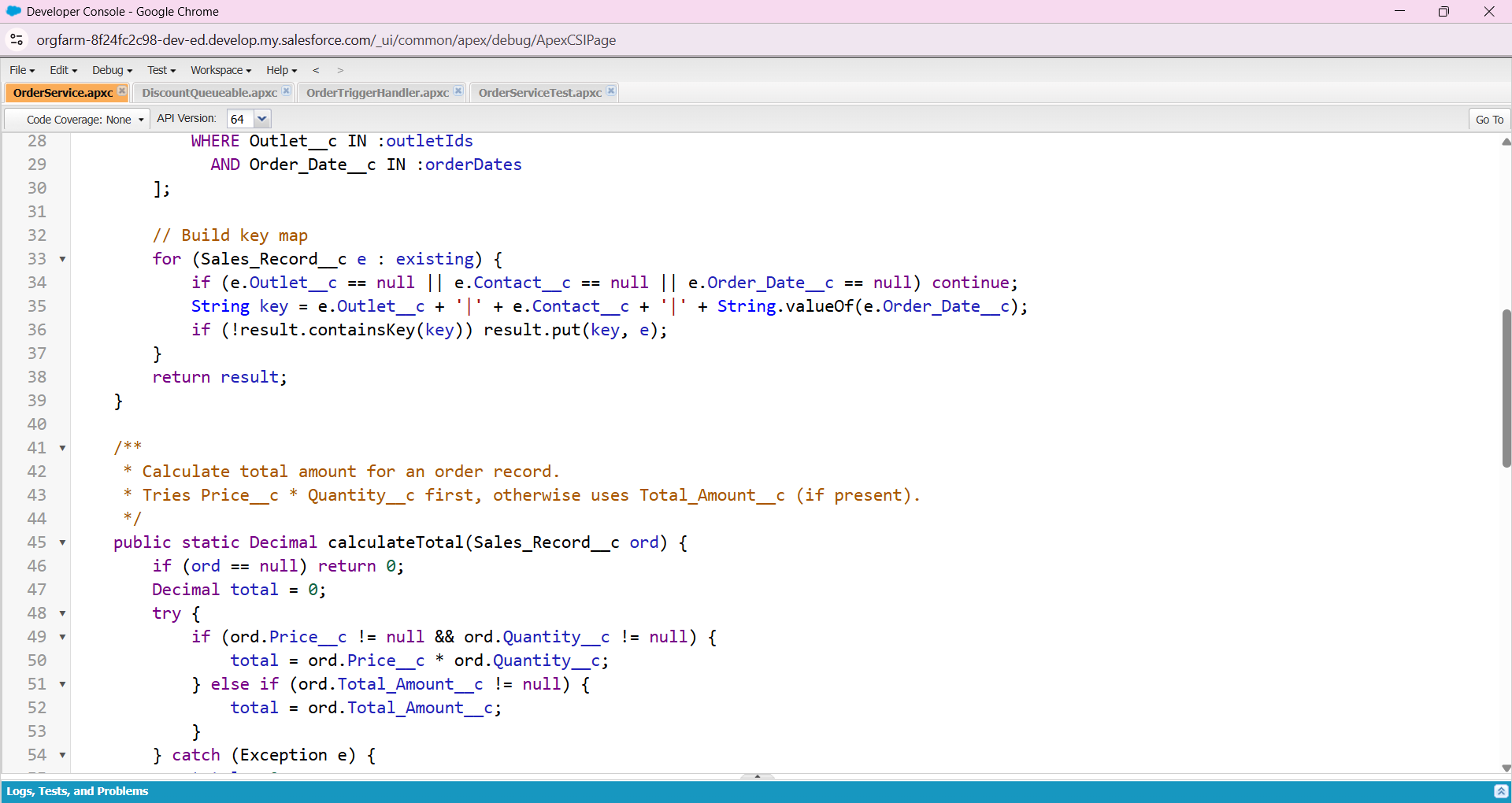
Phase 5 - Salesforce Apex Implementation

# 1. OrderService.cls

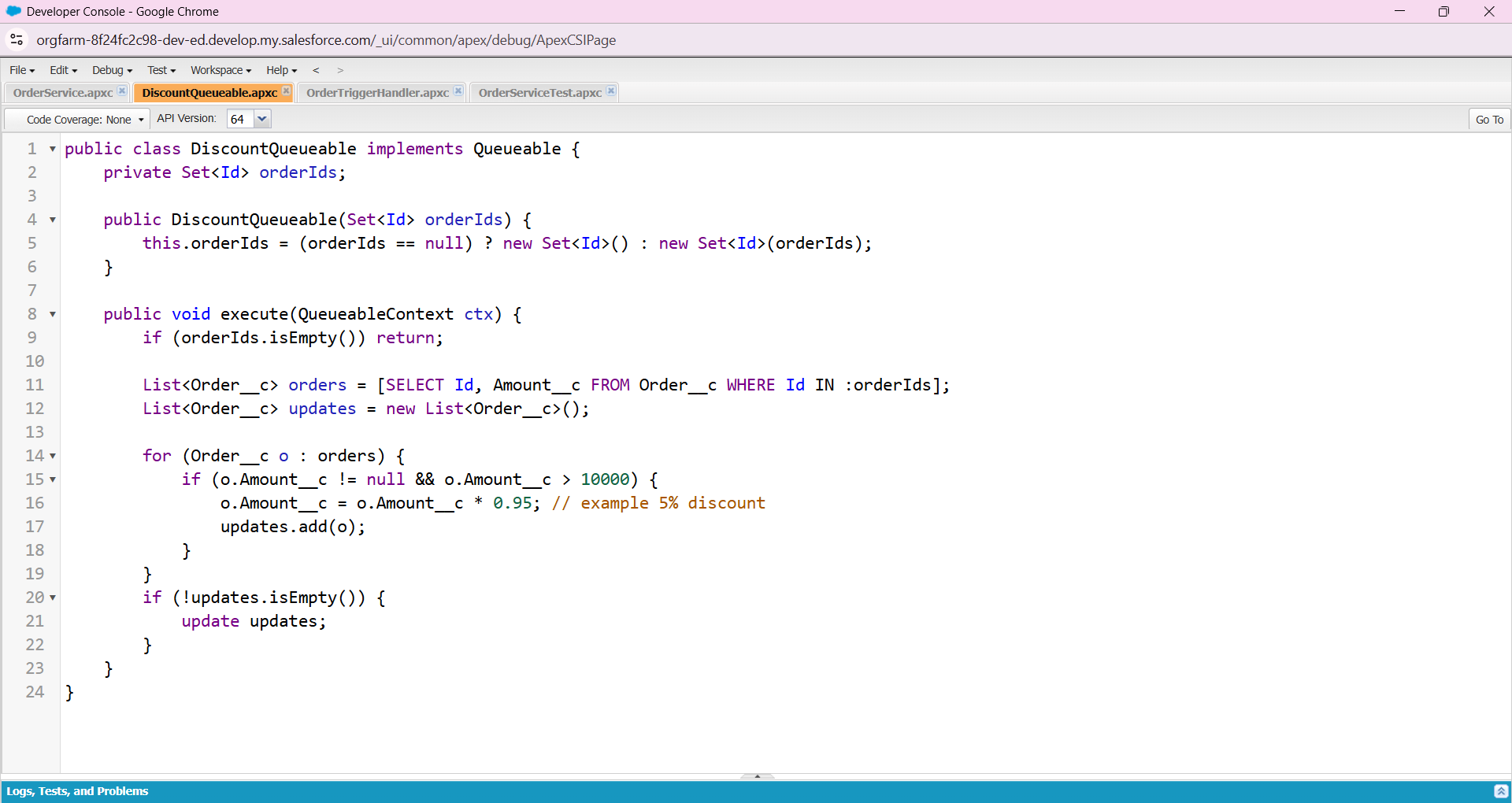
public with sharing class OrderService {  
  
 public static Map<String, Sales\_Record\_\_c> checkForDuplicateOrders(List<Sales\_Record\_\_c> newOrders) {  
 Map<String, Sales\_Record\_\_c> result = new Map<String, Sales\_Record\_\_c>();  
 if (newOrders == null || newOrders.isEmpty()) return result;  
  
 Set<Id> outletIds = new Set<Id>();  
 Set<Date> orderDates = new Set<Date>();  
  
 for (Sales\_Record\_\_c o : newOrders) {  
 if (o.Outlet\_\_c != null && o.Contact\_\_c != null && o.Order\_Date\_\_c != null) {  
 outletIds.add(o.Outlet\_\_c);  
 orderDates.add(o.Order\_Date\_\_c);  
 }  
 }  
  
 if (outletIds.isEmpty() || orderDates.isEmpty()) return result;  
  
 List<Sales\_Record\_\_c> existing = [  
 SELECT Id, Outlet\_\_c, Contact\_\_c, Order\_Date\_\_c  
 FROM Sales\_Record\_\_c  
 WHERE Outlet\_\_c IN :outletIds  
 AND Order\_Date\_\_c IN :orderDates  
 ];  
  
 for (Sales\_Record\_\_c e : existing) {  
 if (e.Outlet\_\_c == null || e.Contact\_\_c == null || e.Order\_Date\_\_c == null) continue;  
 String key = e.Outlet\_\_c + '|' + e.Contact\_\_c + '|' + String.valueOf(e.Order\_Date\_\_c);  
 if (!result.containsKey(key)) result.put(key, e);  
 }  
 return result;  
 }  
  
 public static Decimal calculateTotal(Sales\_Record\_\_c ord) {  
 if (ord == null) return 0;  
 Decimal total = 0;  
 try {  
 if (ord.Price\_\_c != null && ord.Quantity\_\_c != null) {  
 total = ord.Price\_\_c \* ord.Quantity\_\_c;  
 } else if (ord.Total\_Amount\_\_c != null) {  
 total = ord.Total\_Amount\_\_c;  
 }  
 } catch (Exception e) {  
 total = 0;  
 }  
 return total;  
 }  
  
 public static void enqueueDiscountCalculation(Set<Id> orderIds) {  
 if (orderIds == null || orderIds.isEmpty()) return;  
 System.enqueueJob(new DiscountQueueable(orderIds));  
 }  
  
 public static void checkDateConflicts(Sales\_Record\_\_c ord) {  
 if (ord == null) return;  
 if (ord.Outlet\_\_c == null || ord.Start\_Date\_\_c == null || ord.End\_Date\_\_c == null) return;  
  
 List<Sales\_Record\_\_c> overlaps = new List<Sales\_Record\_\_c>();  
 if (ord.Id == null) {  
 overlaps = [  
 SELECT Id, Start\_Date\_\_c, End\_Date\_\_c  
 FROM Sales\_Record\_\_c  
 WHERE Outlet\_\_c = :ord.Outlet\_\_c  
 AND Start\_Date\_\_c <= :ord.End\_Date\_\_c  
 AND End\_Date\_\_c >= :ord.Start\_Date\_\_c  
 LIMIT 1  
 ];  
 } else {  
 overlaps = [  
 SELECT Id, Start\_Date\_\_c, End\_Date\_\_c  
 FROM Sales\_Record\_\_c  
 WHERE Outlet\_\_c = :ord.Outlet\_\_c  
 AND Id != :ord.Id  
 AND Start\_Date\_\_c <= :ord.End\_Date\_\_c  
 AND End\_Date\_\_c >= :ord.Start\_Date\_\_c  
 LIMIT 1  
 ];  
 }  
  
 if (!overlaps.isEmpty()) {  
 ord.addError('This order overlaps with an existing order at the same outlet and cannot be saved.');  
 }  
 }  
}





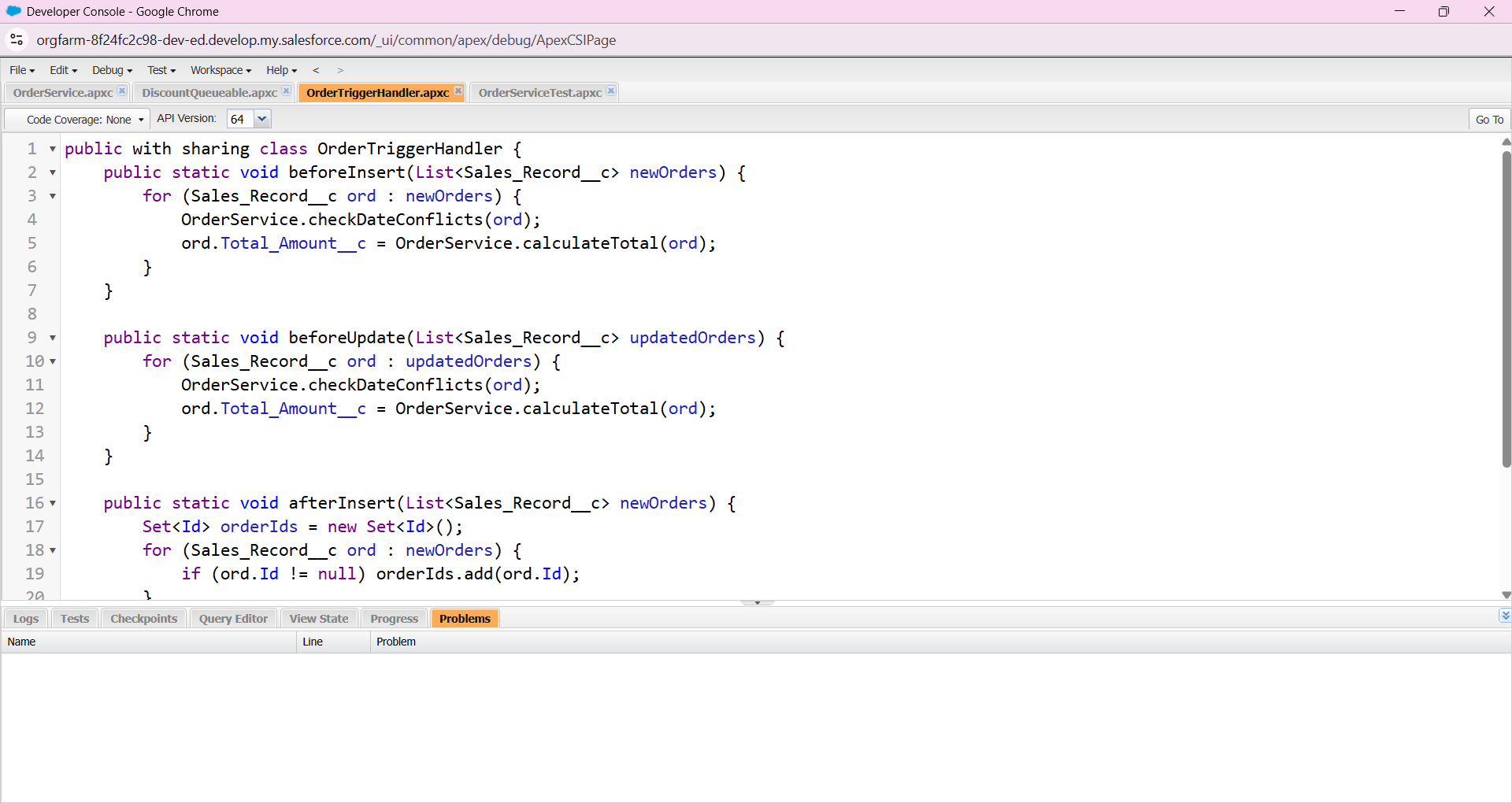
# 2. DiscountQueueable.cls

public class DiscountQueueable implements Queueable {  
 private Set<Id> orderIds;  
  
 public DiscountQueueable(Set<Id> orderIds) {  
 this.orderIds = orderIds;  
 }  
  
 public void execute(QueueableContext context) {  
 List<Sales\_Record\_\_c> orders = [  
 SELECT Id, Total\_Amount\_\_c  
 FROM Sales\_Record\_\_c  
 WHERE Id IN :orderIds  
 ];  
  
 for (Sales\_Record\_\_c ord : orders) {  
 if (ord.Total\_Amount\_\_c != null && ord.Total\_Amount\_\_c > 500) {  
 ord.Total\_Amount\_\_c = ord.Total\_Amount\_\_c \* 0.9;  
 }  
 }  
 if (!orders.isEmpty()) {  
 update orders;  
 }  
 }  
}



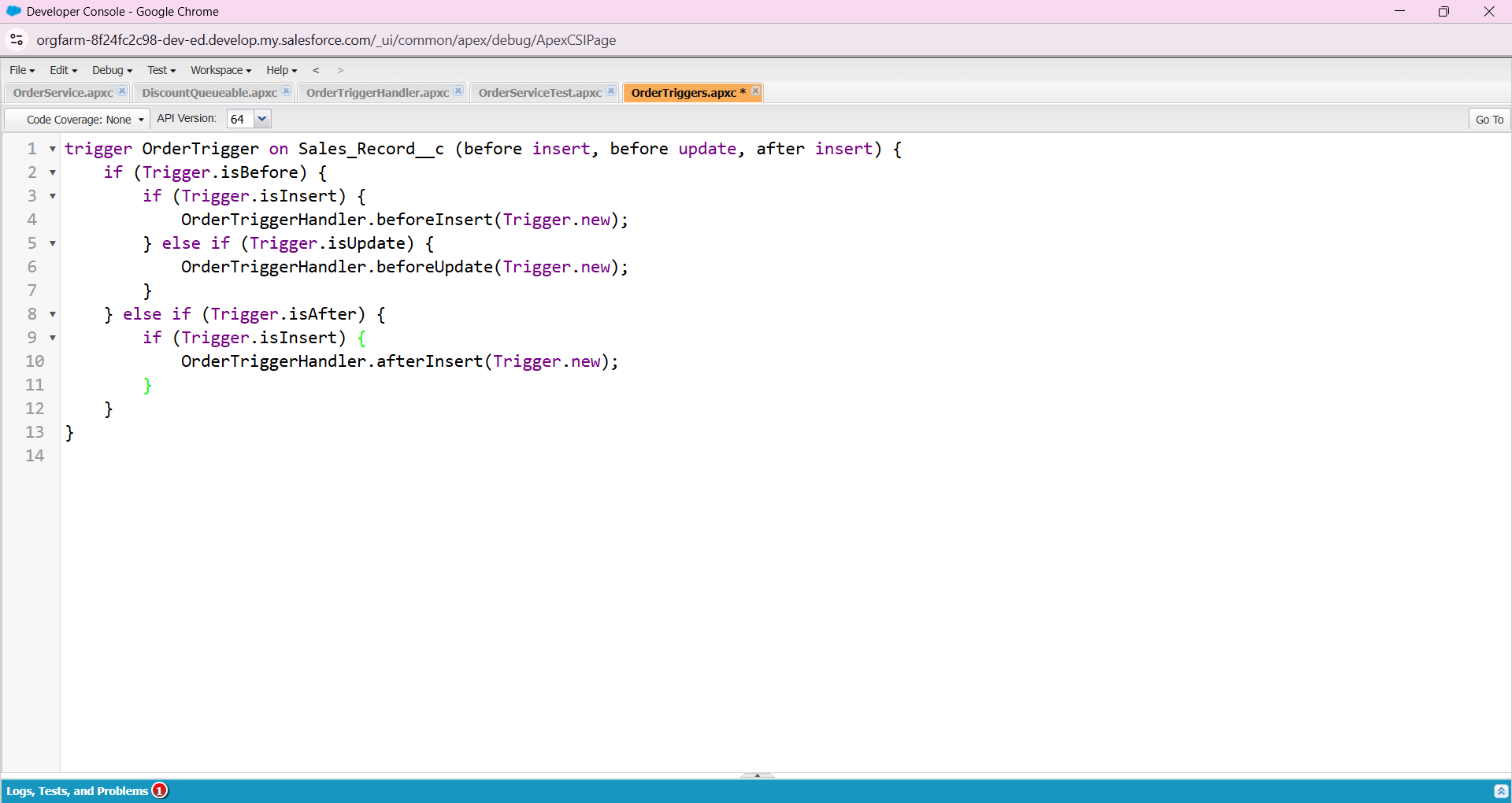
# 3. OrderTriggerHandler.cls

public with sharing class OrderTriggerHandler {  
 public static void beforeInsert(List<Sales\_Record\_\_c> newOrders) {  
 for (Sales\_Record\_\_c ord : newOrders) {  
 OrderService.checkDateConflicts(ord);  
 ord.Total\_Amount\_\_c = OrderService.calculateTotal(ord);  
 }  
 }  
  
 public static void beforeUpdate(List<Sales\_Record\_\_c> updatedOrders) {  
 for (Sales\_Record\_\_c ord : updatedOrders) {  
 OrderService.checkDateConflicts(ord);  
 ord.Total\_Amount\_\_c = OrderService.calculateTotal(ord);  
 }  
 }  
  
 public static void afterInsert(List<Sales\_Record\_\_c> newOrders) {  
 Set<Id> orderIds = new Set<Id>();  
 for (Sales\_Record\_\_c ord : newOrders) {  
 if (ord.Id != null) orderIds.add(ord.Id);  
 }  
 if (!orderIds.isEmpty()) {  
 OrderService.enqueueDiscountCalculation(orderIds);  
 }  
 }  
}



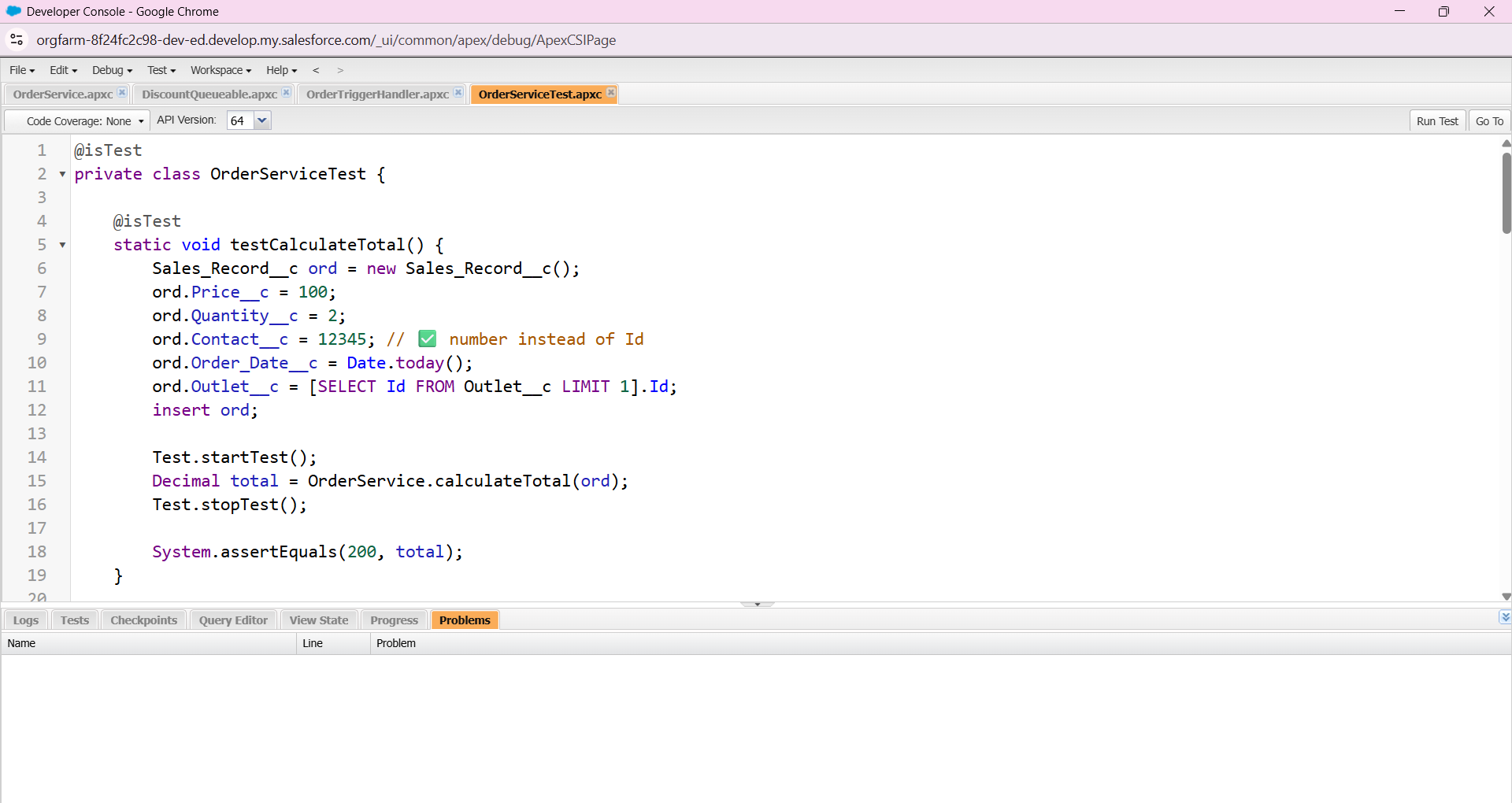
# 4. OrderTrigger.trigger

trigger OrderTrigger on Sales\_Record\_\_c (before insert, before update, after insert) {  
 if (Trigger.isBefore) {  
 if (Trigger.isInsert) {  
 OrderTriggerHandler.beforeInsert(Trigger.new);  
 } else if (Trigger.isUpdate) {  
 OrderTriggerHandler.beforeUpdate(Trigger.new);  
 }  
 } else if (Trigger.isAfter) {  
 if (Trigger.isInsert) {  
 OrderTriggerHandler.afterInsert(Trigger.new);  
 }  
 }  
}



# 5. OrderServiceTest.cls

@isTest  
private class OrderServiceTest {  
  
 @isTest  
 static void testCalculateTotal() {  
 Outlet\_\_c outlet = new Outlet\_\_c(Name='Test Outlet');  
 insert outlet;  
  
 Sales\_Record\_\_c ord = new Sales\_Record\_\_c();  
 ord.Price\_\_c = 100;  
 ord.Quantity\_\_c = 2;  
 ord.Contact\_\_c = 12345;  
 ord.Order\_Date\_\_c = Date.today();  
 ord.Outlet\_\_c = outlet.Id;  
 insert ord;  
  
 Test.startTest();  
 Decimal total = OrderService.calculateTotal(ord);  
 Test.stopTest();  
  
 System.assertEquals(200, total);  
 }  
  
 @isTest  
 static void testCheckForDuplicateOrders() {  
 Outlet\_\_c outlet = new Outlet\_\_c(Name='Duplicate Outlet');  
 insert outlet;  
  
 Sales\_Record\_\_c ord1 = new Sales\_Record\_\_c(  
 Price\_\_c = 50,  
 Quantity\_\_c = 1,  
 Contact\_\_c = 12345,  
 Order\_Date\_\_c = Date.today(),  
 Outlet\_\_c = outlet.Id  
 );  
 insert ord1;  
  
 Sales\_Record\_\_c ord2 = new Sales\_Record\_\_c(  
 Price\_\_c = 50,  
 Quantity\_\_c = 1,  
 Contact\_\_c = 12345,  
 Order\_Date\_\_c = Date.today(),  
 Outlet\_\_c = outlet.Id  
 );  
  
 List<Sales\_Record\_\_c> newOrders = new List<Sales\_Record\_\_c>{ord2};  
  
 Test.startTest();  
 Map<String, Sales\_Record\_\_c> dupes = OrderService.checkForDuplicateOrders(newOrders);  
 Test.stopTest();  
  
 System.assert(!dupes.isEmpty(), 'Duplicate should be found');  
 }  
  
 @isTest  
 static void testCheckDateConflicts() {  
 Outlet\_\_c outlet = new Outlet\_\_c(Name='Conflict Outlet');  
 insert outlet;  
  
 Sales\_Record\_\_c existing = new Sales\_Record\_\_c(  
 Price\_\_c = 10,  
 Quantity\_\_c = 1,  
 Contact\_\_c = 67890,  
 Order\_Date\_\_c = Date.today(),  
 Outlet\_\_c = outlet.Id,  
 Start\_Date\_\_c = Date.today(),  
 End\_Date\_\_c = Date.today().addDays(5)  
 );  
 insert existing;  
  
 Sales\_Record\_\_c newOrder = new Sales\_Record\_\_c(  
 Price\_\_c = 20,  
 Quantity\_\_c = 2,  
 Contact\_\_c = 67890,  
 Order\_Date\_\_c = Date.today(),  
 Outlet\_\_c = outlet.Id,  
 Start\_Date\_\_c = Date.today().addDays(2),  
 End\_Date\_\_c = Date.today().addDays(7)  
 );  
  
 Test.startTest();  
 OrderService.checkDateConflicts(newOrder);  
 Test.stopTest();  
  
 System.assert(newOrder.getErrors().size() > 0, 'Conflict should have been detected');  
 }  
  
 @isTest  
 static void testEnqueueDiscountCalculation() {  
 Outlet\_\_c outlet = new Outlet\_\_c(Name='Discount Outlet');  
 insert outlet;  
  
 Sales\_Record\_\_c ord = new Sales\_Record\_\_c(  
 Price\_\_c = 100,  
 Quantity\_\_c = 1,  
 Contact\_\_c = 99999,  
 Order\_Date\_\_c = Date.today(),  
 Outlet\_\_c = outlet.Id,  
 Start\_Date\_\_c = Date.today(),  
 End\_Date\_\_c = Date.today().addDays(1),  
 Total\_Amount\_\_c = 600  
 );  
 insert ord;  
  
 Test.startTest();  
 OrderService.enqueueDiscountCalculation(new Set<Id>{ord.Id});  
 Test.stopTest();  
  
 System.assert(true, 'Queueable job should have been enqueued');  
 }  
}



### 1. ****OrderService.cls****

This is the main **service class** that holds reusable business logic for Sales Records, like calculating totals, checking duplicates, validating date conflicts, and enqueuing async jobs.

### 2. ****DiscountQueueable.cls****

This is a **Queueable Apex class** that runs discount calculations asynchronously in the background, ensuring bulk operations don’t slow down user transactions.

### 3. ****OrderTriggerHandler.cls****

This is the **trigger handler** class that cleanly organizes logic (before insert, before update, etc.), following the **Trigger Design Pattern** so your trigger remains simple and maintainable.

### 4. ****OrderTrigger.trigger****

This is the **actual trigger** on the Sales\_Record\_\_c object that routes events (insert/update/delete) into the OrderTriggerHandler, making sure your business rules execute automatically when records change.

### 5. ****OrderServiceTest.cls****

This is the **unit test class** that validates the logic in OrderService, ensuring correctness, preventing regressions, and providing >75% code coverage to meet Salesforce deployment requirements.

Bottom of Form