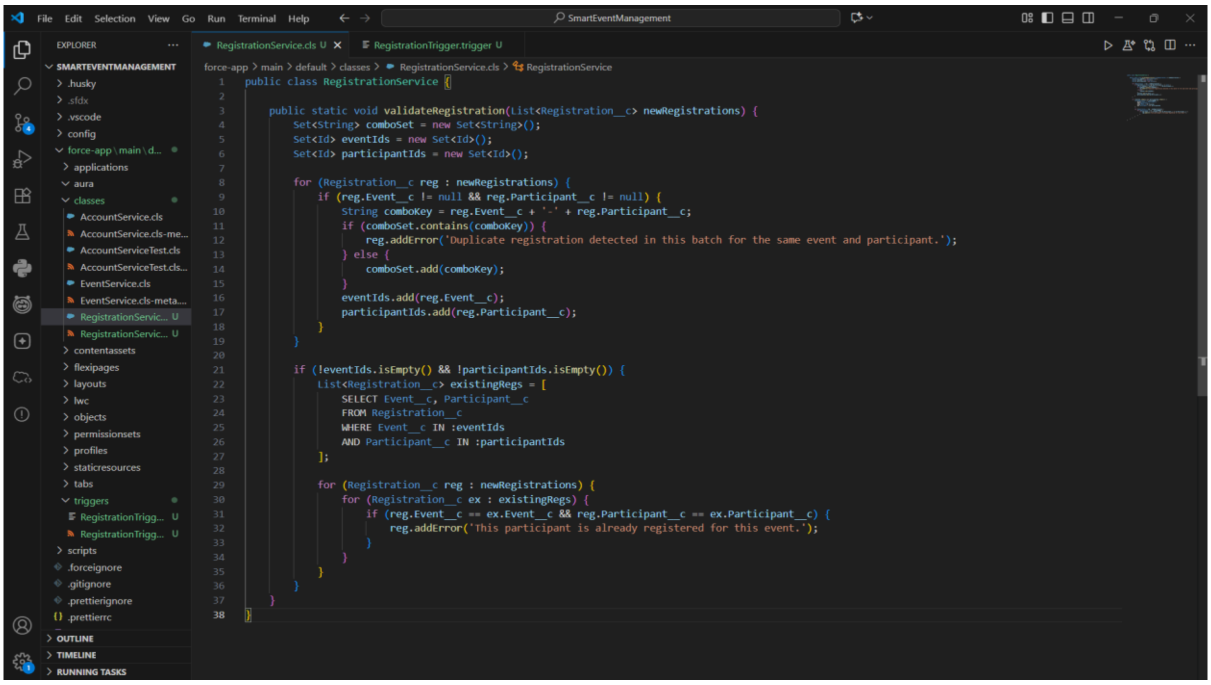
PHASE 5: APEX PROGRAMMING (DEVELOPER)

Goal: Extend the Smart Event Management app with custom Apex logic for validations, automation, asynchronous processing, and unit testing.

Step 1: Apex Classes & Objects

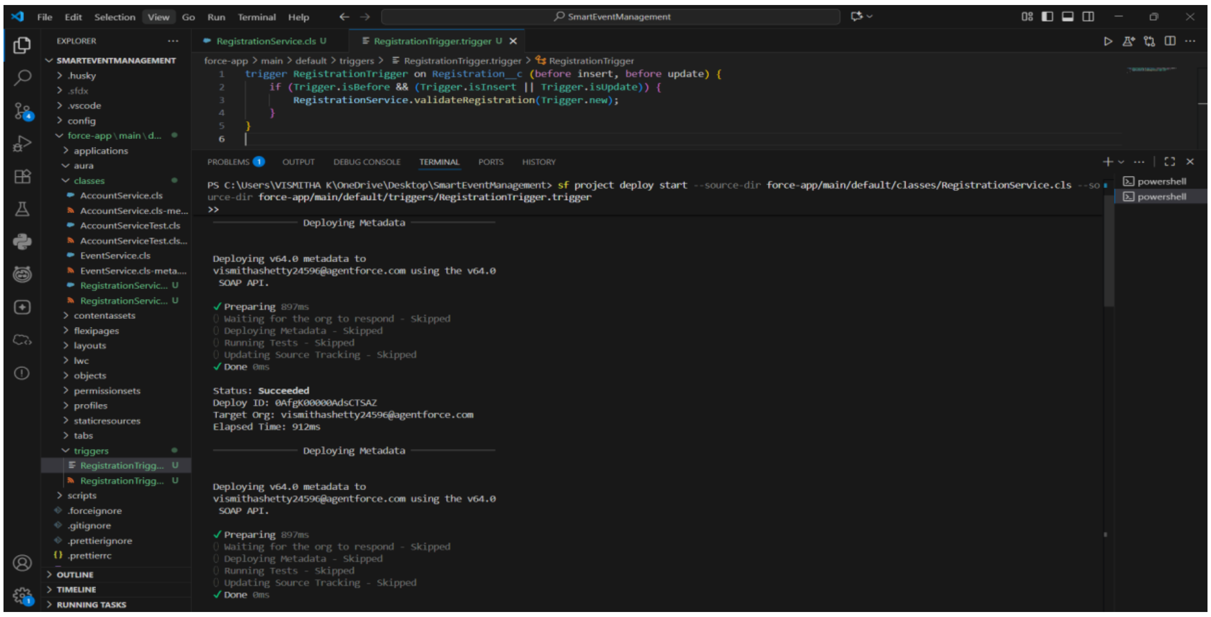
- A new class RegistrationService was created to hold business logic.
- The main responsibility is to validate registrations and prevent duplicates for the same event and participant.
- Logic is kept reusable and centralized, so it can be called from multiple places.
- This ensures the project follows a clean coding pattern where triggers remain lightweight.



```
public class RegistrationService {

    public static void validateRegistration(List<Registration__c> newRegistrations) {
        Set<String> comboSet = new Set<String>();
        Set<Id> eventIds = new Set<Id>();
        Set<Id> participantIds = new Set<Id>();

        for (Registration__c reg : newRegistrations) {
            if (reg.Event__c != null && reg.Participant__c != null) {
                String comboKey = reg.Event__c + '-' + reg.Participant__c;
                if (comboSet.contains(comboKey)) {
                    reg.addError('Duplicate registration detected in this batch for the same event and participant.');
                } else {
                    comboSet.add(comboKey);
                }
                eventIds.add(reg.Event__c);
                participantIds.add(reg.Participant__c);
            }
        }

        if (!eventIds.isEmpty() && !participantIds.isEmpty()) {
            List<Registration__c> existingRegs = [
                SELECT Event__c, Participant__c
                FROM Registration__c
                WHERE Event__c IN :eventIds
                AND Participant__c IN :participantIds
            ];

            for (Registration__c reg : newRegistrations) {
                for (Registration__c ex : existingRegs) {
                    if (reg.Event__c == ex.Event__c && reg.Participant__c == ex.Participant__c) {
                        reg.addError('This participant is already registered for this event.');
                    }
                }
            }
        }
    }
}
```

Step 2: Apex Triggers

- A trigger RegistrationTrigger was implemented on Registration__c.
- It executes before insert and before update, ensuring invalid registrations are blocked before being committed to the database.
- The trigger itself is lean, simply delegating work to the service class.
- This follows the Trigger Handler Design Pattern, which separates trigger context from business logic.

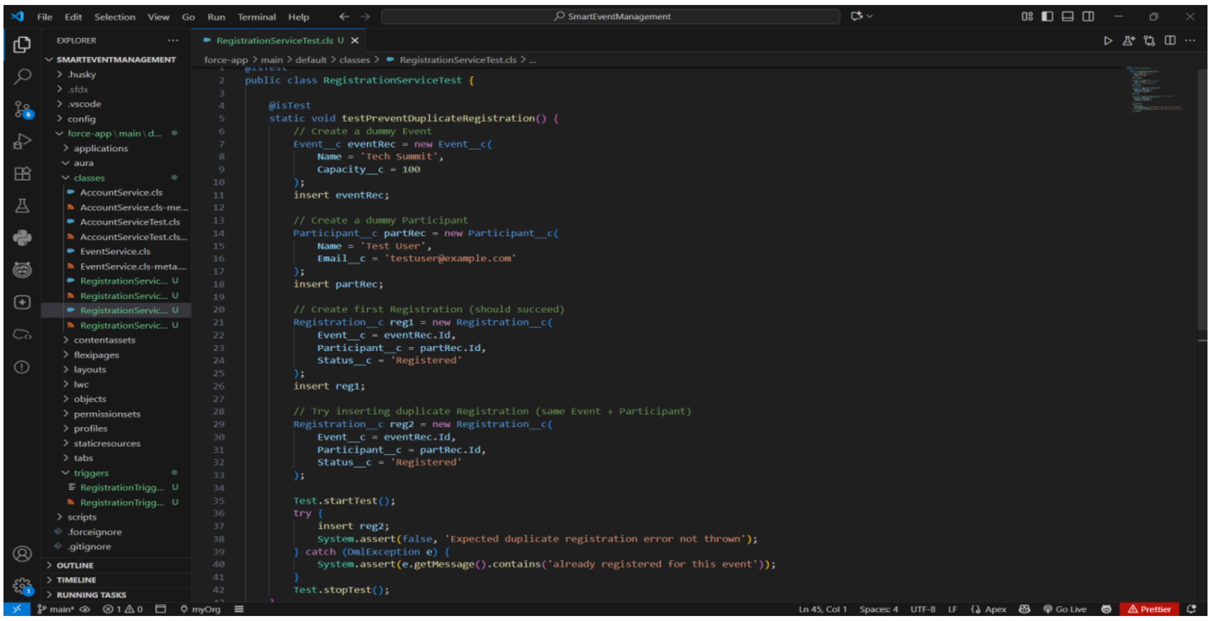


```
trigger RegistrationTrigger on Registration__c (before insert, before update) {
    if (Trigger.isBefore && (Trigger.isInsert || Trigger.isUpdate)) {
        RegistrationService.validateRegistration(Trigger.new);
    }
}
```

```
PS C:\Users\VISMITHA K\OneDrive\Desktop\SmartEventManagement> sf project deploy start --source-dir force-app/main/default/classes/RegistrationService.cls --so
urce-dir force-app/main/default/triggers/RegistrationTrigger.trigger
>>
 ─────────────── Deploying Metadata ───────────────


 Deploying v64.0 metadata to
 vismithashetty24596@agentforce.com using the v64.0
  SOAP API.

 ✓ Preparing 897ms
 ◯ Waiting for the org to respond - Skipped
 ◯ Deploying Metadata - Skipped
 ◯ Running Tests - Skipped
 ◯ Updating Source Tracking - Skipped
 ✓ Done 0ms

 Status: Succeeded
 Deploy ID: 0AfgK00000AdsCTSAZ
 Target Org: vismithashetty24596@agentforce.com
 Elapsed Time: 912ms

 ─────────────── Deploying Metadata ───────────────


 Deploying v64.0 metadata to
 vismithashetty24596@agentforce.com using the v64.0
  SOAP API.

 ✓ Preparing 897ms
 ◯ Waiting for the org to respond - Skipped
 ◯ Deploying Metadata - Skipped
 ◯ Running Tests - Skipped
 ◯ Updating Source Tracking - Skipped
 ✓ Done 0ms
```

Step 3: Test Classes

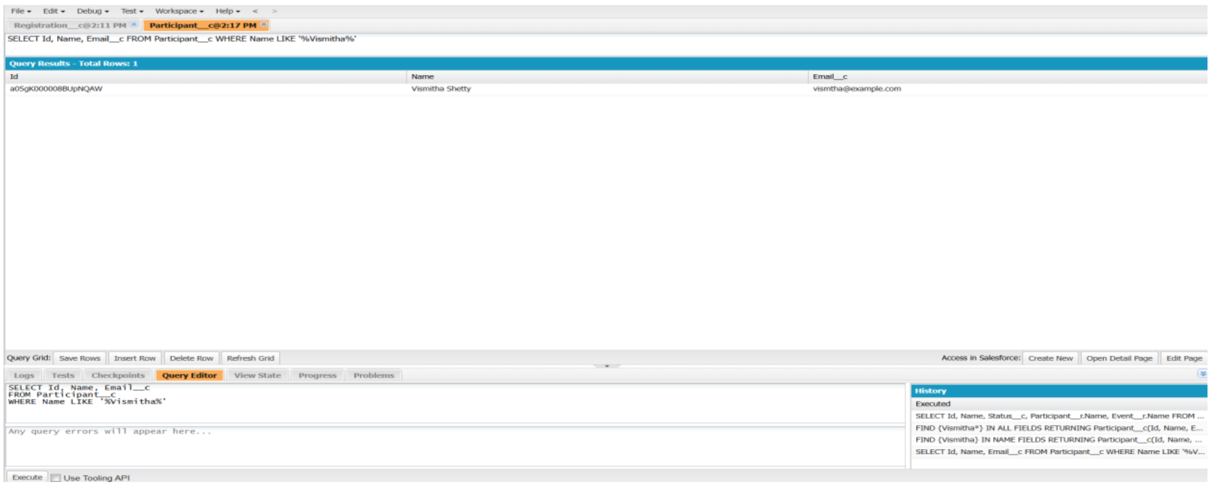
- A dedicated test class was created to verify the registration validation logic.
- Positive scenario: a participant can successfully register for an event once.
- Negative scenario: duplicate registration is blocked with a meaningful error.
- Assertions confirm that the logic works as intended.
- This test class also ensures code coverage requirements are met for deployment.
- An initial failure occurred due to a conflicting Flow; this was resolved by bypassing/deactivating it during test runs.



```
public class RegistrationServiceTest {

    @isTest
    static void testPreventDuplicateRegistration() {
        // Create a dummy Event
        Event__c eventRec = new Event__c(
            Name = 'Tech Summit',
            Capacity__c = 100
        );
        insert eventRec;

        // Create a dummy Participant
        Participant__c partRec = new Participant__c(
            Name = 'Test User',
            Email__c = 'testuser@example.com'
        );
        insert partRec;

        // Create first Registration (should succeed)
        Registration__c reg1 = new Registration__c(
            Event__c = eventRec.Id,
            Participant__c = partRec.Id,
            Status__c = 'Registered'
        );
        insert reg1;

        // Try inserting duplicate Registration (same Event + Participant)
        Registration__c reg2 = new Registration__c(
            Event__c = eventRec.Id,
            Participant__c = partRec.Id,
            Status__c = 'Registered'
        );

        Test.startTest();
        try {
            insert reg2;
            System.assert(false, 'Expected duplicate registration error not thrown');
        } catch (DmlException e) {
            System.assert(e.getMessage().contains('already registered for this event'));
        }
        Test.stopTest();
```

Step 4: SOQL & SOSL

- SOQL was applied to fetch specific records with conditions (e.g., all registrations of an event, or participants by partial name).
- SOSL was explored as a global search tool across objects. While SOSL executed without errors, for custom objects like Participant__c it did not return expected results consistently. Therefore, SOQL was chosen as the reliable method in this project.
- This reflects a real-world scenario where developers choose between SOQL and SOSL depending on object type and search requirements.


File Edit Debug Test Workspace Help
Registration__c@2:11 PM
Participant__c@2:17 PM
Query Grid: Save Rows Insert Row Delete Row Refresh Grid
Access in Salesforce: Create New Open Detail Page Edit Page
Logs Tests Checkpoints Query Editor View State Progress Problems
Execute
Use Tooling API

```
SELECT Id, Name, Email__c FROM Participant__c WHERE Name LIKE '%Vismitha%'
```

Query Results - Total Rows: 1

Id	Name	Email__c
a05gK000008BUpNQAW	Vismitha Shetty	vismtha@example.com

```
SELECT Id, Name, Email__c
FROM Participant__c
WHERE Name LIKE '%Vismitha%'
```

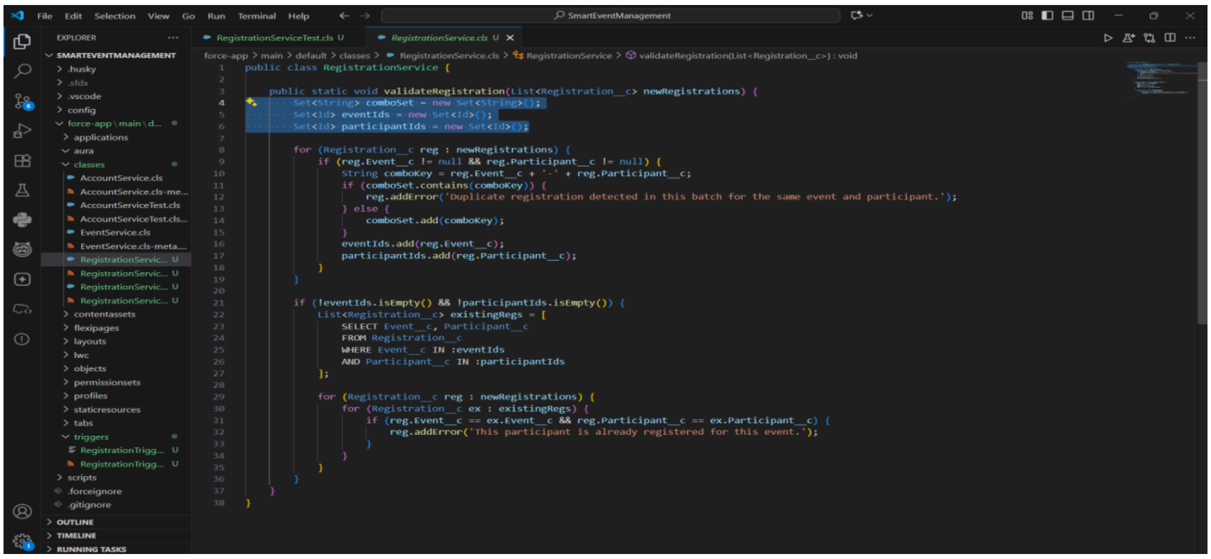
```
Any query errors will appear here...
```

History
Executed

```
SELECT Id, Name, Status__c, Participant__r.Name, Event__r.Name FROM ...
FIND {Vismitha*} IN ALL FIELDS RETURNING Participant__c(Id, Name, E...
FIND {Vismitha} IN NAME FIELDS RETURNING Participant__c(Id, Name, ...
SELECT Id, Name, Email__c FROM Participant__c WHERE Name LIKE '%V...
```

Step 5: Collections (List, Set, Map)

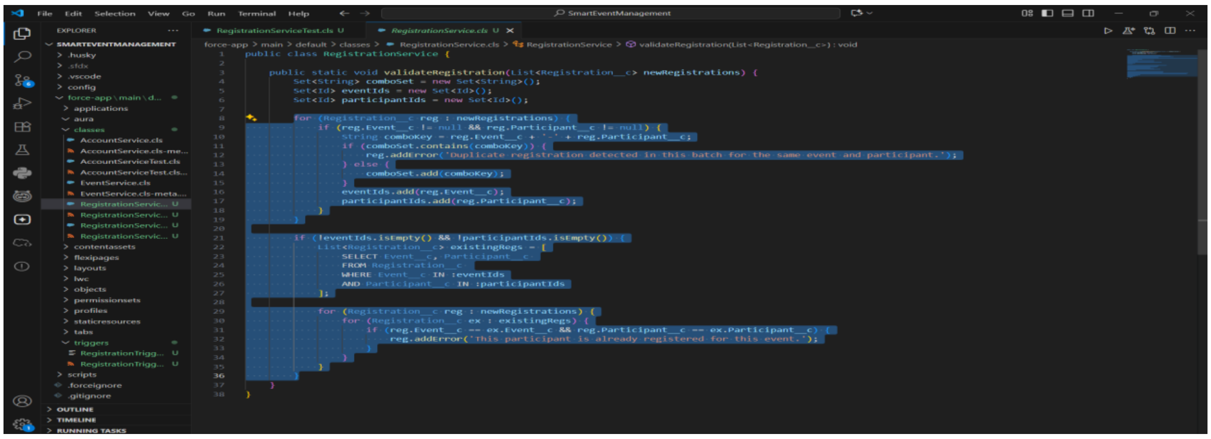
- Collections were used extensively to handle data efficiently.
- **Lists** stored multiple Registration records for processing.
- **Sets** ensured uniqueness of Event–Participant pairs, preventing duplicate entries.
- **Maps** allowed quick lookups, such as linking Event Ids to Event records.
- These data structures optimized both performance and logic clarity.


File Edit Selection View Go Run Terminal Help
SmartEventManagement
EXPLORER
SMARTEVENTMANAGEMENT
RegistrationServiceTest.cls U
RegistrationService.cls U
force-app > main > default > classes > RegistrationService.cls > RegistrationService > validateRegistration(List<Registration__c>) : void
OUTLINE
TIMELINE
RUNNING TASKS

```
public class RegistrationService {

    public static void validateRegistration(List<Registration__c> newRegistrations) {
        Set<String> comboSet = new Set<String>();
        Set<Id> eventIds = new Set<Id>();
        Set<Id> participantIds = new Set<Id>();

        for (Registration__c reg : newRegistrations) {
            if (reg.Event__c != null && reg.Participant__c != null) {
                String comboKey = reg.Event__c + '-' + reg.Participant__c;
                if (comboSet.contains(comboKey)) {
                    reg.addError('Duplicate registration detected in this batch for the same event and participant.');
                } else {
                    comboSet.add(comboKey);
                }
                eventIds.add(reg.Event__c);
                participantIds.add(reg.Participant__c);
            }
        }

        if (!eventIds.isEmpty() && !participantIds.isEmpty()) {
            List<Registration__c> existingRegs = [
                SELECT Event__c, Participant__c
                FROM Registration__c
                WHERE Event__c IN :eventIds
                AND Participant__c IN :participantIds
            ];

            for (Registration__c reg : newRegistrations) {
                for (Registration__c ex : existingRegs) {
                    if (reg.Event__c == ex.Event__c && reg.Participant__c == ex.Participant__c) {
                        reg.addError('This participant is already registered for this event.');
                    }
                }
            }
        }
    }
}
```

Step 6: Control Statements

- **If-else** conditions determined when to block registrations.
- **Loops** processed batches of records in trigger context.
- Decision-making through control statements ensures rules are applied consistently across all scenarios.



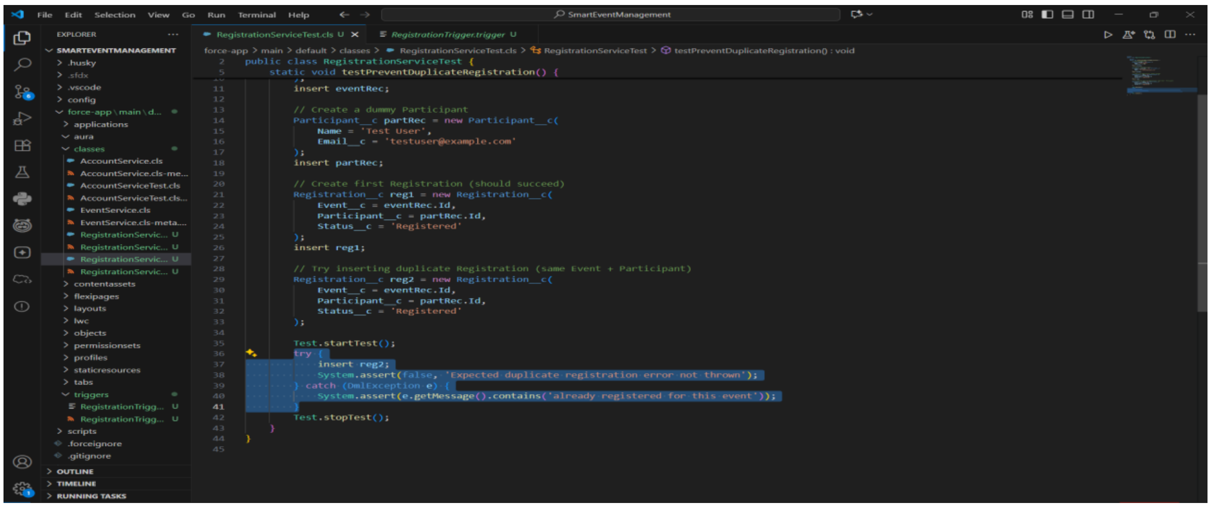
```
public class RegistrationService {

    public static void validateRegistration(List<Registration__c> newRegistrations) {
        Set<String> comboSet = new Set<String>();
        Set<Id> eventIds = new Set<Id>();
        Set<Id> participantIds = new Set<Id>();

        for (Registration__c reg : newRegistrations) {
            if (reg.Event__c != null && reg.Participant__c != null) {
                String comboKey = reg.Event__c + '-' + reg.Participant__c;
                if (comboSet.contains(comboKey)) {
                    reg.addError('Duplicate registration detected in this batch for the same event and participant.');
                } else {
                    comboSet.add(comboKey);
                }
                eventIds.add(reg.Event__c);
                participantIds.add(reg.Participant__c);
            }
        }

        if (!eventIds.isEmpty() && !participantIds.isEmpty()) {
            List<Registration__c> existingRegs = [
                SELECT Event__c, Participant__c
                FROM Registration__c
                WHERE Event__c IN :eventIds
                AND Participant__c IN :participantIds
            ];

            for (Registration__c reg : newRegistrations) {
                for (Registration__c ex : existingRegs) {
                    if (reg.Event__c == ex.Event__c && reg.Participant__c == ex.Participant__c) {
                        reg.addError('This participant is already registered for this event.');
                    }
                }
            }
        }
    }
}
```

Step 7: Asynchronous Apex

- **Future methods** were used to handle lightweight asynchronous operations, such as sending confirmation data to external systems without delaying the user.
- **Queueable Apex** was explored to manage bulk operations like applying discounts across large sets of registrations. Its ability to chain jobs makes it more powerful than future methods.
- **Batch Apex** was implemented to process large volumes of registrations in chunks, for example marking waitlisted registrations as expired after an event has ended.
- **Scheduled Apex** was set up to automate routine tasks such as sending daily event summary reports to managers. The job runs automatically at predefined times, demonstrating Salesforce's ability to support enterprise automation.
- **Asynchronous Apex** ensures scalability, efficiency, and better system performance, especially for business processes that require heavy data processing or external integrations.

Step 8: Exception Handling

- Applied to handle errors gracefully and provide user-friendly messages.
- Business rule violations are surfaced to users with clear messages (e.g., duplicate registration).
- System errors are caught and logged for debugging without breaking end-user experience.
- Example usage in project is shown below. Similarly exceptions are handled in other required places too.



```
public class RegistrationServiceTest {
    static void testPreventDuplicateRegistration() {
        insert eventRec;

        // Create a dummy Participant
        Participant__c partRec = new Participant__c(
            Name = 'Test User',
            Email__c = 'testuser@example.com'
        );
        insert partRec;

        // Create first Registration (should succeed)
        Registration__c reg1 = new Registration__c(
            Event__c = eventRec.Id,
            Participant__c = partRec.Id,
            Status__c = 'Registered'
        );
        insert reg1;

        // Try inserting duplicate Registration (same Event + Participant)
        Registration__c reg2 = new Registration__c(
            Event__c = eventRec.Id,
            Participant__c = partRec.Id,
            Status__c = 'Registered'
        );

        Test.startTest();
        try {
            insert reg2;
            System.assert(false, 'Expected duplicate registration error not thrown');
        } catch (DmlException e) {
            System.assert(e.getMessage().contains('already registered for this event'));
        }
        Test.stopTest();
    }
}
```

Step 9: Asynchronous Processing — Project Use Cases

- Asynchronous processing was applied to simulate real-world use cases in the Smart Event Management system.
- **Batch Apex** was designed to automatically expire waitlisted registrations after the event has ended. This ensures registration data remains consistent and up to date without manual intervention.
- **Queueable Apex** was implemented to handle sponsor-driven discounts for participants. When triggered, the system bulk-updates registration fees, making the process efficient and scalable.
- **Future Methods** were used to integrate with external systems for communication. For example, once a participant registers, their details can be sent to an SMS/email service without delaying the registration save.
- **Scheduled Apex** was used to provide daily event summaries automatically to Event Managers, ensuring they have the latest updates without running reports manually.

ACHIEVEMENTS IN PHASE 5

- Enforced critical business rules with custom Apex code.
- Created a modular service class with a supporting trigger.
- Built and executed test classes for validation and code coverage.
- Applied SOQL, SOSL, collections, and control statements effectively.
- Understood and documented the use of asynchronous Apex techniques for scalability.
- Ensured robust exception handling to maintain system stability.