Phase 5: Apex Programming (Developer)

- In Phase 5, I focused on Apex programming concepts in Salesforce to implement backend business logic, automate workflows, and handle asynchronous processing in the Recruitment Management System (RMS) project. Below are the concepts I implemented, along with practical scenarios from the project.

1. Classes & Objects

❖ **Explanation:** In Apex, classes are templates that define objects, their attributes, and methods. Objects are instances of classes. Classes help in **organizing code, improving reusability, and implementing business logic.**

❖ **Scenario:**

- Created JobPostHandler class to manage operations related to job posts, such as validating inputs, auto-generating job codes, and updating the status.
- Created ApplicationHandler class to manage candidate applications, including eligibility checks, status updates, and notifications.
- Example: When a recruiter creates a new job post, the JobPostHandler class validates the data and assigns default recruiters if none are selected.

2. Apex Triggers (Before/After Insert/Update/Delete)

❖ **Explanation:** Triggers automatically execute before or after DML operations (insert, update, delete) to enforce business rules or automate processes.

❖ **Scenario:**

- **Before Insert:** Prevented duplicate applications for the same candidate and job.
- **After Insert:** Sent notifications to HR and recruiters about new applications.
- **Before Update:** Validated that job post deadlines are not in the past.
- **After Update:** Created tasks for interview scheduling when application status changes.
- **Before Delete:** Blocked deletion of applications that are already shortlisted.
  Example: Application Prevent Duplicate Trigger

```
public class ApplicationTriggerHandler {

   // Prevent duplicate applications
   public static void
preventDuplicateApplications(List<Application__c>
newApps) {
      Set<Id> contactIds = new Set<Id>();
      Set<Id> jobIds = new Set<Id>();
      for(Application__c app : newApps){
         if(app.Contact__c != null)
contactIds.add(app.Contact__c);
         if(app.JobPost1__c != null)
jobIds.add(app.JobPost1__c);
      }
      List<Application__c> existingApps = [SELECT Id,
Contact__c, JobPost1__c
                                 FROM Application__c
                                 WHERE Contact__c IN
:contactIds
                                 AND JobPost1__c IN :jobIds];
      Set<String> existingKeys = new Set<String>();
```

```
        for(Application__c app : existingApps){
            existingKeys.add(app.Contact__c + '-' +
app.JobPost1__c);
        }
        for(Application__c app : newApps){
            String key = app.Contact__c + '-' +
app.JobPost1__c;
            if(existingKeys.contains(key)){
                app.addError('Candidate has already applied for
this job.');
            }
        }
    }

    // Create task when status changes
    public static void
createTasksOnStatusChange(List<Application__c>
newApps, Map<Id, Application__c> oldMap){
        List<Task> tasksToCreate = new List<Task>();
        for(Application__c app : newApps){
            Application__c oldApp = oldMap.get(app.Id);
            if(oldApp.Status__c != app.Status__c &&
               app.Status__c == 'Shortlisted' &&
               app.Assigned_User__c != null){
                Task t = new Task();
                t.Subject = 'Follow up on shortlisted application';
                t.WhatId = app.Id;
                t.OwnerId = app.Assigned_User__c;
                t.Status = 'Not Started';
                t.Priority = 'High';
                t.Description = 'Application approved. Follow
up with candidate.';
                tasksToCreate.add(t);
            }
        }
```

```
        if(!tasksToCreate.isEmpty()) insert tasksToCreate;
    }
}
```

ApplicationTrigger:

```
trigger ApplicationTrigger on Application__c (before
insert, after update) {
    if(Trigger.isBefore && Trigger.isInsert){

ApplicationTriggerHandler.preventDuplicateApplicat
ions(Trigger.new);
    }
    if(Trigger.isAfter && Trigger.isUpdate){

ApplicationTriggerHandler.createTasksOnStatusCha
nge(Trigger.new, Trigger.oldMap);
    }
}
```

3. Trigger Design Pattern

❖ **Explanation:**

The Trigger Design Pattern separates business logic from triggers into handler classes, making code more reusable, maintainable, and testable.

❖ **Scenario:**

- Created **ApplicationTriggerHandler** and **JobPostTriggerHandler** classes.
- Triggers only call handler methods instead of containing logic themselves.
- Example: When a candidate's application status changes to "shortlisted," a task is automatically created for the recruiter.

Job Post Handler:

```
public class JobPostHandler {


    // Validate Job Post before insert
    public static void validateJobPosts(List<JobPost__c> newJobs){
        for(JobPost__c job : newJobs){
            if(job.Last_Date__c < Date.today()){
                job.addError('Job closing date cannot be in the past.');
            }
        }
    }
}
```

Job Post Trigger:

```
trigger JobPostTrigger on JobPost__c (before insert, before update) {
    if(Trigger.isBefore){
        JobPostHandler.validateJobPosts(Trigger.new);
    }
}
```

4. SOQL & SOSL

❖ **Explanation:**

- **SOQL (Salesforce Object Query Language)**: Fetches records from objects with filters.

- **SOSL (Salesforce Object Search Language):**
  Performs text-based searches across multiple objects.

❖ **Scenario:**

- **SOQL:** Fetch all applications for a candidate:

```
List<Application__c> apps = [SELECT Id,
Applicant_Status__c
                    FROM Application__c
                    WHERE Contact__c = :candidateId];
```

- **SOSL:** Quickly search candidates by name, email, or phone across multiple objects.

5.Collections: List, Set, Map

❖ **Explanation:**
Collections store multiple records efficiently:

- **List:** Ordered, allows duplicates.
- **Set:** Unordered, unique values.
- **Map:** Key-value pairs for fast lookups.

❖ **Scenario:**

- **List:** Store all applications pending review.
- **Set:** Store unique candidate emails to prevent duplicates.
- **Map:** Map Application Id → Job Id for bulk updates.

6. Control Statements

❖ **Explanation:**
Used for decision-making and loops (if-else, for, while, switch).

❖ **Scenario:**
Assign recruiters based on job category:

```
if(job.Category__c == 'Tech') {
        job.Assigned_Recruiter__c = techRecruiterId;
```

```
} else if(job.Category__c == 'HR') {
        job.Assigned_Recruiter__c = hrRecruiterId;
} else {
        job.Assigned_Recruiter__c = defaultRecruiterId;
}
```

7. Exception Handling

❖ **Explanation:**
Try-catch blocks are used to handle runtime exceptions gracefully.

❖ **Scenario:**

When sending notifications via Apex, exceptions are caught to prevent transaction failure:

```
try {
   Messaging.sendEmail(new
List<Messaging.SingleEmailMessage>{email});
} catch(Exception e) {
   System.debug('Email sending failed: ' + e.getMessage());
}
```

8. Test Classes

❖ **Explanation:**
Test classes ensure Apex code works correctly and meet **75% code coverage** required for deployment.

❖ **Scenario:**

- Tested creation of valid applications, prevention of duplicates, and automatic task creation.
- Example: **TestApplicationHandler** class verifies all application workflows:

```
@IsTest
public class TestApplicationTrigger {
    static testMethod void testDuplicatePrevention() {
        JobPost__c job = new JobPost__c(Name='Developer',
Status__c='Open', Last_Date__c=Date.today().addDays(10));
        insert job;
        Contact candidate = new Contact(FirstName='John',
LastName='Doe');
        insert candidate;

        Application__c app1 = new
Application__c(Contact__c=candidate.Id, JobPost__c=job.Id,
Status__c='Applied');
        insert app1;

        Application__c app2 = new
Application__c(Contact__c=candidate.Id, JobPost__c=job.Id,
Status__c='Applied');

        Test.startTest();
        try {
            insert app2;
        } catch(DmlException e) {
            System.assert(e.getMessage().contains('already
applied'));
        }
        Test.stopTest();
    }
}
```

9. Asynchronous Processing

❖ **Explanation:**
Used for operations that run in the background without blocking the main transaction.

❖ **Scenario:**

- **Batch Apex:** Close expired job postings automatically.
- **Queueable Apex:** Send mass notifications for new job postings.
- **Scheduled Apex:** Remind recruiters about upcoming interviews.
- **Future Methods:** Integrate with external candidate verification systems.

```
@future
public static void sendCandidateVerification(Id
candidateId){
   // Callout to external verification service
}
```