

Tasks

- Automatically created tasks for recruiters when interview is scheduled, ensuring follow-ups.

Custom Notifications

- Real-time notifications to users inside Salesforce for important events such as:
 - Interview scheduled, mode of notifications are emails.
 - Application approved or rejected.

.

Phase 5: Apex Programming (Developer)

In Phase 5, I focused on **Apex programming concepts** in Salesforce to add backend business logic, automation,

and asynchronous processing to the Job Portal project. Below are the details of the concepts I implemented along with scenarios.

1. Classes & Objects

Explanation:

In Apex, classes are templates that define objects, their attributes, and methods. Objects are instances of classes. They help in organizing code, applying reusability, and implementing business logic.

Scenario:

I created an Apex class `Job Application Handler` to manage operations related to job applications, such as validating applicant details and assigning interviewers. For example, when a new applicant record is created, the class methods are used to check eligibility before saving.

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

Explanation:

Triggers are used to perform actions automatically before or after DML (Data Manipulation Language) operations like insert, update, or delete.

Scenario:

- **Before Insert:** Prevented duplicate job applications for the same position by the same candidate.
- **After Insert:** Sent an automatic notification to HR after a job application was submitted.
- **Application Prevent duplicate Handler this** apex trigger helps in preventing duplicate records of applicant for the same contact and job opening.

```
public class Application_Trigger_Handler {  
  
    public static void preventDuplicateApplications(List  
newApps) {  
  
        // Collect all Contact and Job Ids from  
the incoming records  
        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.Job__c != null) {  
                jobIds.add(app.Job__c);  
            }  
        }  
    }  
}
```

```

    // Query existing Applications with
those with Contact and Job combinations
    List<Application__c> existingApps = [
        SELECT Id, Contact__c, Job__c
        FROM Application__c
        WHERE Contact__c IN :contactIds
        AND Job__c IN :jobIds
    ];

    // Build a set of existing keys
(ContactId + JobId)
    Set<String> existingKeys = new
Set<String>();
    for (Application__c app : existingApps) {
        existingKeys.add(app.Contact__c + '-'
+ app.Job__c);
    }

    // Compare with new records → block
duplicates
    for (Application__c app : newApps) {
        String key = app.Contact__c + '-' +
app.Job__c;
        if (existingKeys.contains(key)) {
            app.addError('This candidate has
already applied for this job posting.');
```

```
}
```

```
}
```

Application Prevent Duplicate Trigger

trigger Applicaiton_Trigger on Application__c (before insert)

```
{ if (Trigger.isBefore && Trigger.isInsert)
{ Application_Trigger_Handler.preventDuplicateApplications(Trigger.new); }
}
```

3. Trigger Design Pattern

Explanation:

The Trigger Design Pattern ensures that triggers are clean, scalable, and maintainable. Business logic is separated into handler classes instead of writing directly inside the trigger.

Scenario:

For the Application__c object, instead of writing all logic inside the trigger, I created ApplicationTriggerHandler class which handled validations, notifications, and updates. The trigger simply called the handler methods, making it reusable and cleaner.

- **Create Application from contact created and Existing Job Opening** - this creates application automatically when a contact associated with a job opening is being created.

```
public class Application_Trigger_Handler_1 {  
  
    // Method to create Applications from  
    Contacts who applied  
  
    public static void  
    createApplicationsFromContacts(List<Contact>  
    newContacts) {  
  
        List<Application__c> appsToCreate = new  
        List<Application__c>();  
  
        //: Loop through Contacts  
        for (Contact c : newContacts) {
```

```

        // Only create Application if
Job_Posting__c is filled
        if (c.Job_Opening__c != null) {

            // : Prevent duplicate
Application for same Contact + Job
            List<Application__c> existingApps
= [
                SELECT Id FROM Application__c
                WHERE Contact__c = :c.Id
                AND      Job__c
= :c.Job_Opening__c
            ];
            if (existingApps.isEmpty()) {
                Application__c app = new
Application__c();
                app.Contact__c = c.Id;
                app.Job__c =
c.Job_Opening__c;
                app.Applicant_Status__c =
'Applied';
                appsToCreate.add(app);
            }
        }

// Insert Applications
if (!appsToCreate.isEmpty()) {
    insert appsToCreate;
}

```

```
    }  
}  
  
}
```

- **Application Status Handler** -> whenever the application status is updated to shortlisted then a task is created and is assigned to the recruiter who will be taking the interview as a notification about the interview .

```
public class Application_Status_Trigger_Hander {  
  
    // Method to create Task when Application  
    status changes  
    public static void  
    createTaskOnStatusChange(List<Application__c>  
    newApps, Map<Id, Application__c> oldMap) {  
  
        List<Task> tasksToCreate = new  
        List<Task>();  
  
        for (Application__c app : newApps) {  
  
            // Compare old vs new status to  
            detect change  
            Application__c oldApp =
```



```

oldMap.get(app.Id);

        if (oldApp.Applicant_Status__c !=
app.Applicant_Status__c &&
app.Applicant_Status__c == 'shortlisted' &&
app.Assigned_User__c != null) {

            Task t = new Task();
            t.Subject = 'Follow up on
shortlisted Application';
            t.WhatId = app.Id; // Related to
Application
            t.OwnerId = app.Assigned_User__c;
// Assign to recruiter (replace with your
field API name)
            t.Status = 'Not Started';
            t.Priority = 'High';
            t.Description = 'The application
has been approved. Follow up with the
candidate.';
            tasksToCreate.add(t);
        }
    }

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

```

```
}
```

Application status Trigger

```
trigger Application_status_trigger on Application__c (after  
update) {
```

```
// Call handler method, pass Trigger.new and  
Trigger.oldMap
```

```
Application_Status_Trigger_Hander.createTaskOnStatusC  
hange(Trigger.new, Trigger.oldMap); }
```

Contact trigger

```
trigger Contact_Trigger_1 on Contact (after insert, after  
update) {
```

```
List<Contact> contactsWithJob = new  
List<Contact>();
```

```
// Step 1: Loop through inserted/updated  
contacts
```

```
for (Contact c : Trigger.new) {  
    if (c.Job_Opening__c != null) { //  
replace with your actual field API name  
        contactsWithJob.add(c);  
    }  
}
```

```
}  
  
// Step 2: Call handler to create  
Applications  
if (!contactsWithJob.isEmpty()) {  
  
Application_Trigger_Handler_1.createApplicati  
onsFromContacts(contactsWithJob);  
}  
  
}
```

4. SOQL & SOSL

Explanation:

- **SOQL (Salesforce Object Query Language):** Used to fetch records from Salesforce objects based on conditions.
- **SOSL (Salesforce Object Search Language):** Used to perform text-based searches across multiple objects.

Scenario:

- SOQL was used to fetch all applications for a given candidate (SELECT Id, Status FROM Application__c WHERE Candidate__c = :candidateId).
- SOSL was used to search applicant details (like email/phone) across objects when HR wanted to quickly find a candidate.

5. Collections: List, Set, Map

Explanation:

Collections are data structures used to store multiple records.

- **List:** Ordered collection allowing duplicates.
- **Set:** Unordered collection without duplicates.
- **Map:** Key-value pairs for quick lookups.

Scenario:

- **List:** Used to store all interview records for a particular application.
- **Set:** Used to store unique candidate emails to prevent duplicates.
- **Map:** Used to map Application Id → Interview Date for quick access in bulk processing.

6. Control Statements

Explanation:

Control statements like if-else, for, while, and switch are used to apply decision-making and looping logic.

Scenario:

When assigning an interviewer, I used control statements:

- If the application status is "Interview Scheduled", then assign an interviewer.
- Else if the status is "Rejected", mark the application as closed.

12. Test Classes

Explanation:

Test classes are written to verify that Apex code works correctly and to meet Salesforce's requirement of 75% code coverage for deployment.

Scenario:

For each trigger and class, I wrote test classes such as TestApplicationHandler which tested:

- Creating a valid application

- Preventing duplicate applications
- Scheduling interviews

This ensured that all logic worked as expected before deployment.

```
@IsTest
public class ATS_TestClass {

    // Utility method to create a Contact
    private static Contact createContact() {
        Contact c = new Contact(
            LastName = 'Test Candidate',
            Email = 'testcandidate@example.com'
        );
        insert c;
        return c;
    }

    // Utility method to create a Job Posting
    private static Job_Opening__c createJobPosting() {
        Job_Opening__c job = new Job_Opening__c(
            Name = 'Software Engineer'
        );
        insert job;
        return job;
    }

    // Utility method to create an Application
    private static Application__c createApplication(Id contactId, Id jobId,
String statusVal) {
        Application__c app = new Application__c(
            Contact__c = contactId,
            Job__c = jobId,
            Applicant_Status__c = statusVal
        );
        insert app;
        return app;
    }

    // Utility method to create Applicant Info + Interview
    private static Interview__c createInterview(Id appInfoId, Datetime
slotTime) {
        Interview__c interview = new Interview__c(
            Applicant_Information__c = appInfoId,
```

```

        Interview_Date_Time__c = slotTime
    );
    insert interview;
    return interview;
}

// -----
// TEST CASES
// -----

@IsTest
static void testDuplicateApplicationPrevention() {
    Contact c = createContact();
    Job_Opening__c job = createJobPosting();

    // Insert first application
    Application__c app1 = createApplication(c.Id, job.Id, 'Applied');

    // Try inserting duplicate application
    Application__c app2 = new Application__c(
        Contact__c = c.Id,
        Job__c = job.Id,
        Applicant_Status__c = 'Applied'
    );

    Test.startTest();
    try {
        insert app2;
        System.assert(false, 'Duplicate should not be inserted');
    } catch (DmlException e) {
        System.assert(e.getMessage().contains('duplicate'), 'Should block
duplicate application');
    }
    Test.stopTest();
}

@IsTest
static void testTaskCreationOnApprovedApplication() {
    Contact c = createContact();
    Job_Opening__c job = createJobPosting();

    Application__c app = createApplication(c.Id, job.Id, 'Applied');

    Test.startTest();
    app.Applicant_Status__c = 'Approved';
    update app;
}

```

```

        Test.stopTest();

        // Check Task created
        List<Task> tasks = [SELECT Id, Subject, WhatId FROM Task WHERE WhatId
= :app.Id];
        System.assertEquals(1, tasks.size(), 'Task should be created when
Application is Approved');
        System.assertEquals('Application Approved Notification',
tasks[0].Subject, 'Task subject should match');
    }

    @IsTest
    static void testInterviewValidation_NoOverlap() {
        Contact c = createContact();
        Job_Opening__c job = createJobPosting();
        Application__c app = createApplication(c.Id, job.Id, 'Applied');

        // First interview slot
        Interview__c int1 = createInterview(app.Id,
Datetime.now().addDays(1));

        // Overlapping interview slot
        Interview__c int2 = new Interview__c(
            Applicant_Information__c = app.Id,
            Interview_Date_Time__c = int1.Interview_Date_Time__c // same time
        );

        Test.startTest();
        try {
            insert int2;
            System.assert(false, 'Should not allow overlapping interviews');
        } catch (DmlException e) {
            System.assert(e.getMessage().contains('overlap'), 'Should block
overlapping interview creation');
        }
        Test.stopTest();
    }
}

```


13. Asynchronous Processing

Explanation:

Asynchronous processing (Batch Apex, Queueable, Scheduled, Future methods) allows operations to run in the background without blocking the main execution.

Scenario:

- Batch Apex: Closing inactive applications.
- Queueable Apex: Sending notifications for new job postings.
- Scheduled Apex: Interview reminders.
- Future Method: Background verification with external systems.

This ensured better performance and scalability of the system.

Phase 6: User Interface Development

In this phase, the focus was on creating an intuitive and interactive interface for recruiters and HR managers to manage Job Postings, Applications, and Interviews efficiently. Salesforce Lightning Experience along with **Lightning Web Components (LWC)** was leveraged to enhance usability, display dynamic data, and integrate backend functionality.