

# Phase 5: Apex Programming (Developer)

## 1. Classes & Objects

- LeaveRequestController: Handles leave application logic (submit, update, approve, reject).
- Utility Classes: For reusable logic like date validation, string formatting, and error handling.

## 2. Apex Triggers

### Before Insert/Update:

- Validate leave dates (From\_Date ≤ To\_Date).
- Prevent overlapping leave requests.

### After Insert:

- Notify manager of new leave request submission.

### After Update:

- If Leave is approved change the status of the leave whether it is approved.
- If Leave is rejected change the status of the leave whether it is rejected.

<input type="checkbox"/>	A0002	2023-03-19	2023-03-19	For personal reason	Rejected	Edit
<input type="checkbox"/>	A0001	2023-03-15	2023-03-15	Test	Pending	Edit

- If Leave is in pending change the status of the leave whether it is Pending.

My Leaves    Leave Requests							
<input type="checkbox"/>	Request Id	From Date	To Date	Reason	Status	Manager Comment	
<input type="checkbox"/>	A0012	2025-09-25	2025-09-27	health issue	Pending		Edit
<input type="checkbox"/>	A0011	2025-09-26	2025-09-30	health sick	Pending		Edit
<input type="checkbox"/>	A0010	2025-09-26	2025-09-29	wetest	Pending		Edit
<input type="checkbox"/>	A0009	2025-09-26	2025-09-29	webtest	Pending		Edit

## 3. Trigger Design Pattern

### Handler Class Pattern followed:

- One trigger per object (OnsubmitHandler).
- Delegates logic to OnsubmitHandler class.
- Improves readability, reusability, and testability.

## 4. SOQL & SOSL Usage:

SOQL (Salesforce Object Query Language) is used to fetch records from Salesforce objects. In your code:

```
<lightning-datatable key-field="Id" data={myLeaves} columns={columns}  
onrowaction={rowActionHandler}></lightning-datatable>
```

- **data={myLeaves}** → This property is populated by a **JavaScript @wire or Apex method** that fetches leave request records.
- Example Apex method:

```
@AuraEnabled(cacheable=true)  
public static List<LeaveRequest__c> getUserLeaves(Id userId) {  
    return [  
        SELECT Id, From_Date__c, To_Date__c, Status__c, Reason__c  
        FROM LeaveRequest__c  
        WHERE User__c = :userId  
        ORDER BY From_Date__c DESC  
    ];  
}
```

- This SOQL query fetches all leave requests for the current user (`User__c = :userId`) to display in the datatable.
- The results populate `myLeaves` in the LWC and show each record in a row.

## SOSL :

**SOSL (Salesforce Object Search Language)** is used for **searching across multiple objects or fields**, typically with a search term.

- In your current template, there is **no direct SOSL usage**.
- SOSL would be relevant if you had a **search bar** and wanted to find leave requests by keyword in fields like **Reason\_\_c**, **Status\_\_c**, or **Employee Name**.

Example SOSL in Apex :

```
@AuraEnabled(cacheable=true)

public static List<LeaveRequest__c> searchLeaves(String searchTerm) {

    List<List<SObject>> searchResults = [FIND :searchTerm IN ALL FIELDS RETURNING LeaveRequest__c(Id,
    From_Date__c, To_Date__c, Reason__c, Status__c)];

    return (List<LeaveRequest__c>)searchResults[0];

}
```

- Searches LeaveRequest\_\_c records across all fields for the term entered by the user.
- Useful for a global search feature in your LWC.

### 3. Record Creation / Update:

```
<lightning-record-edit-form object-api-name={objectApiName} record-id={recordId}
onsuccess={successHandler} onsubmit={submitHandler}>

    <lightning-input-field field-name="User__c" value={currentUserId}></lightning-input-field>

    <lightning-input-field field-name="From_Date__c"></lightning-input-field>

    <lightning-input-field field-name="To_Date__c"></lightning-input-field>

    <lightning-input-field field-name="Reason__c"></lightning-input-field>

</lightning-record-edit-form>
```

#### What it does:

- Allows the user to **create or edit** a Leave Request record.
- record-id={recordId} → If empty, creates a new record; if filled, updates an existing record.
- onsuccess={successHandler} → Called after save to refresh data.

#### Relation to SOQL/SOSL:

- Not direct, but **under the hood**, Salesforce uses **SOQL** to fetch field data for the record when editing.
- Upon submission, Salesforce performs a **DML operation** (insert/update) on the object.

## Control Statements:

If-Else: Approve vs Reject logic.

Checks if **From\_Date\_\_c > To\_Date\_\_c**.

If true, shows an **error toast** → “From date should not be greater than To date.”

Prevents invalid date ranges.

Compares today's date (new Date()) with the From Date.

If From Date is **earlier than today**, shows an **error toast** → “From date should not be less than Today.”

If both validations pass:

- Submits the form with updated fields (including Status\_\_c = 'Pending').
- Uses this.refs.leaveRequestFrom (reference to the form) to perform the save operation.

```
• if (new Date(fields.From_Date__c)>new Date(fields.To_Date__c)){  
•     this.ShowToast('From date should not be grater than to  
date','Error','error');  
•     }  
•     else if(new Date()>new Date(fields.From_Date__c)){  
•         this.ShowToast('From date should not be less than  
Today','Error','error');  
•     }  
•     else{  
•         this.refs.leaveRequestFrom.submit(fields);  
•     }
```

## 5. Asynchronous Apex:

Automatically runs when the component loads or when reactive parameters change.

Salesforce handles caching and re-fetching.

You don't write .then() or await; data comes through the result.

Best for read-only data that should auto-refresh (like a list of leave requests in a table).

```

@wire(getMyLeaves)
wiredMyLeaves(result) {
  this.myLeavesWireResult = result;
  if (result.data) {
    this.myLeaves = result.data.map(a => ({
      ...a,
      cellClass: a.Status__c === 'Approved'
        ? 'slds-theme_success'
        : a.Status__c === 'Rejected'
        ? 'slds-theme_warning'
        : "",
      isEditDisabled: a.Status__c !== 'Pending'
    }));
  }
  if (result.error) {
    console.error('Error occurred while fetching my leaves: ', result.error);
  }
}

```

## 6.Exception Handling

- Try-Catch Blocks: Handle DML and SOQL exceptions.

```

• try {
•   update leaveRecord;
• } catch(DmlException e) {
•   System.debug('Error: ' + e.getMessage());
• }

```

- Custom Exceptions: For business rules like “Insufficient Leave Balance.”
- Error Logging: Store errors in a custom object Error\_Log\_\_c for admin review.