# Customer 360 Banking – Capstone Project Report

*A Salesforce-based CRM solution for the Banking Industry*

## Table of Contents



## Phase 1: Problem Understanding & Industry Analysis

*Goal: To thoroughly understand the challenges faced by the banking sector and identify opportunities for a CRM solution.*

1. **Requirement Gathering**

Talk to stakeholders (Relationship Managers, Compliance Officers, Bank Management, IT).

Example requirements:
■ Unified Customer Profile (accounts, loans, cards, investments).
■ Automated loan application & approval flow with KYC check.
■ Transaction visibility & repayment tracking.
■ Audit-ready reports for compliance (KYC/AML).
■ Real-time notifications for high-risk/fraud events.
■ Dashboards for branch performance and customer segmentation.

2. **Stakeholder Analysis**

○ Admin (project setup, system configuration).
○ Relationship Manager (view full customer 360, propose products).
○ Branch Manager (approve loans, view branch dashboards).
○ Compliance Officer / Auditor (access read-only reports, KYC verification).
○ Customer Support (handle service cases, update case status).
○ Customer (experience faster service, view status via portal if implemented).

3. **Business Process Mapping**

- Map current (as-is) flows: separate systems for deposits, loans, cards, support → manual handoffs & data re-entry.
- Target (to-be) flows: single-entry into Salesforce → auto-routing, approvals, and updates to a centralized Customer 360.
- Example process: Customer applies for loan → RM creates Loan application → System triggers credit/KYC check → If below threshold auto-approve, else goes to Branch Manager → Disbursement & EMI schedule created → Transactions posted to Loan record.

4. **Industry-specific Use Case Analysis**

Customer onboarding (KYC collection, verification, account creation).
Loan origination (application → credit check → approval → disbursement).
Repayment tracking (EMIs, overdue detection).
Product cross-sell (recommend investment/credit card based on profile).
Fraud detection (unusual transactions, rapid big transfers).

5. **AppExchange Exploration**

- Evaluate AppExchange apps for: KYC/document verification, credit bureau integration, document e-signature (DocuSign), and secure file storage.
- Decide which to integrate later vs build in-house for learning value.

**Phase Output / Next Steps:** A clear understanding of the banking industry's pain points and a preliminary assessment of how Salesforce can address them. Proceed to solution design.

## Phase 2: Org Setup and Configuration

*Goal: Prepare Salesforce environment to model banking processes, security, and users.*

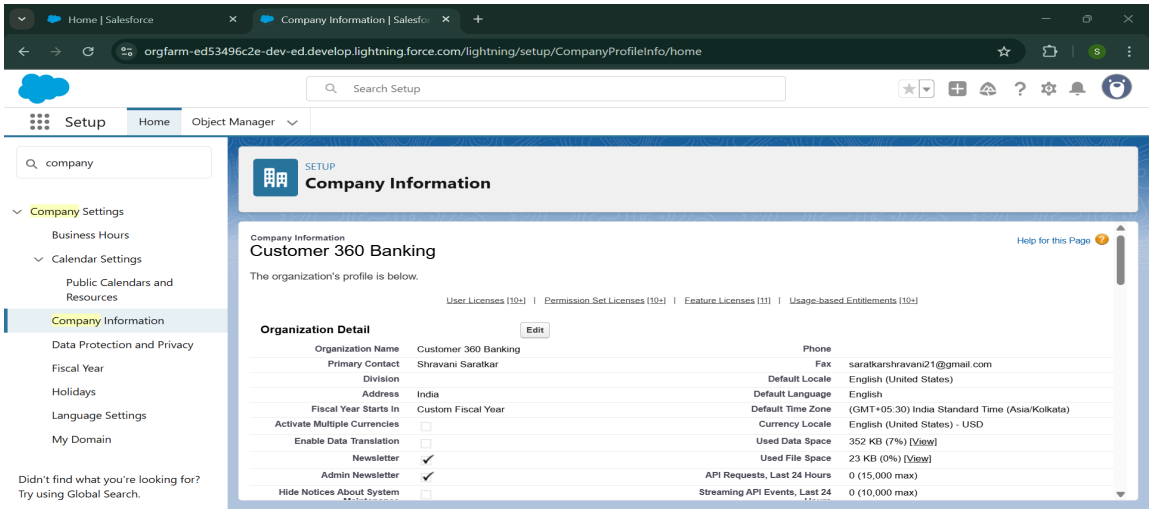
1. Salesforce Editions

- Use Developer Edition for building and testing (suitable for capstone).
  Note: Enterprise or Financial Services Cloud is recommended for production features.

2. Company Profile Setup

  ○ Company Name: Customer 360 Banking.
  ○ Default Currency: USD (set for global-friendly demos).

○ Default Time Zone & Locale: Asia/Kolkata (or org preference).




3. Business Hours & Holidays

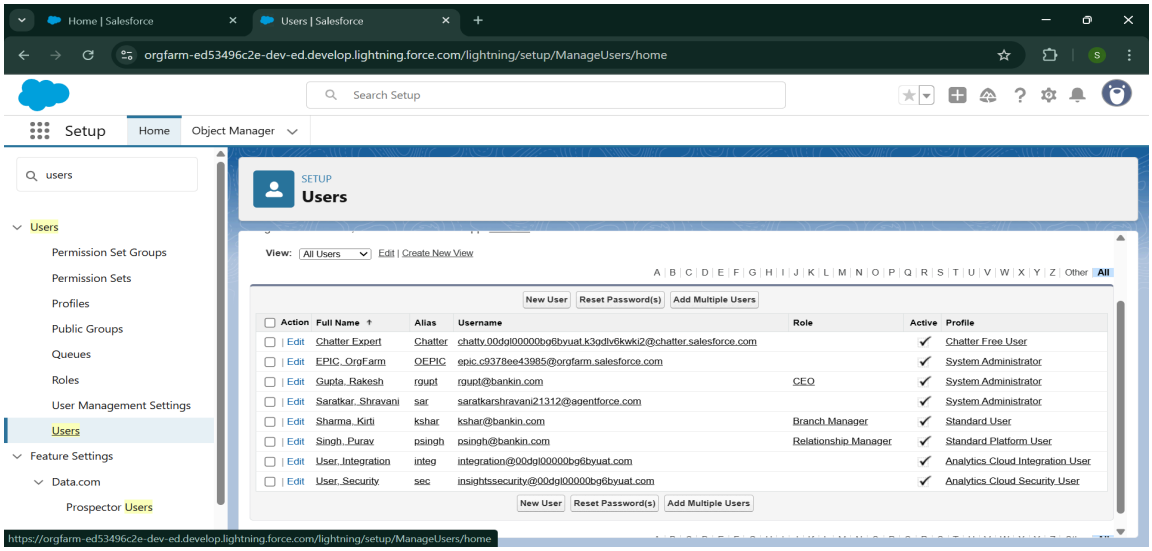
○ Configure Business Hours: Mon–Fri, 09:00 AM – 06:00 PM (IST). Weekends excluded.
○ Mark the Business Hours record Active and Use as Default.

4. Fiscal Year Settings

○ Enabled Custom Fiscal Year aligned to April → March (FY 2025–2026 example).
○ Template: Gregorian Calendar (12 months/year) and set start date to 01-Apr-2025 if modelling Indian banking cycle.

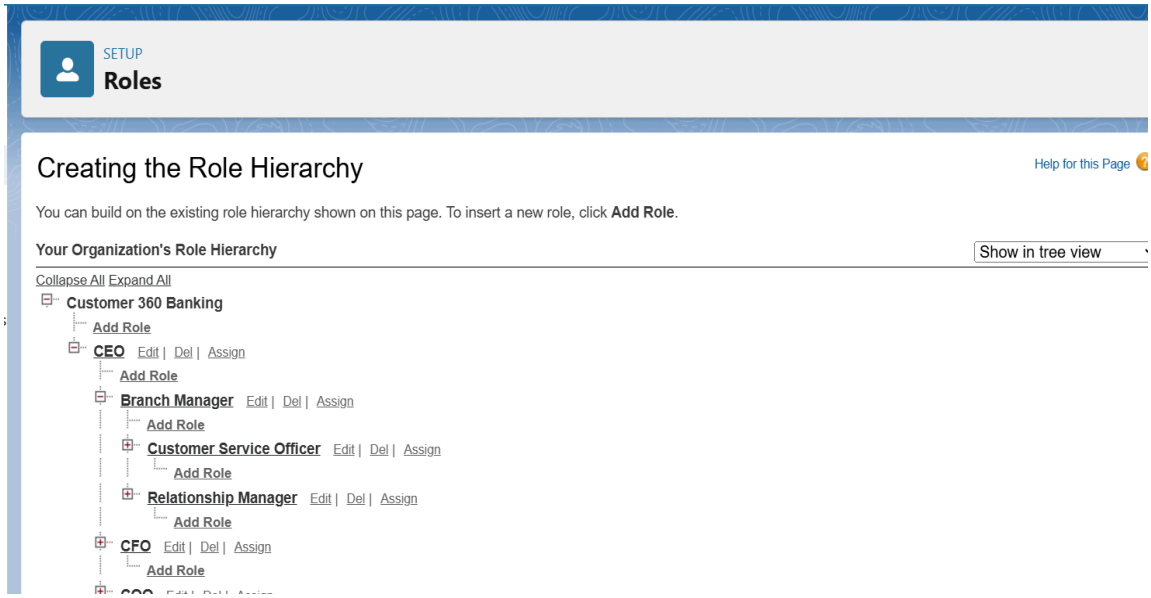
5. User Setup & Licenses

○ Created sample users (use email+alias pattern for unique usernames):
    ■ Admin — System Administrator (full access).
    ■ Branch Manager — Manager role (approval authority).
    ■ Relationship Manager — front-line RM (create loan requests).
    ■ Customer Support Officer — case handling.
○ Optionally add Compliance Officer (read-only access) or Portal/Community users later.




6. Profiles & Roles

○ Profiles: Use System Administrator for Admin; clone Standard User → "Bank Staff Profile" for RM/Support if customization needed.
○ Role Hierarchy example: CEO (top) → Branch Manager → Relationship Manager → Customer Support. This enables record visibility roll-up.




7. Permission Sets

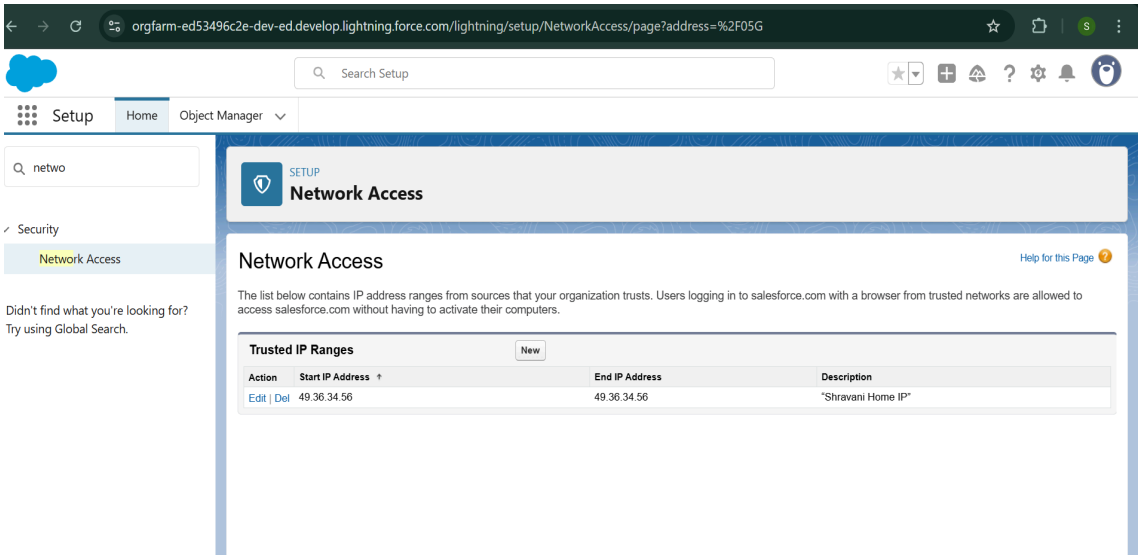
○ Create Permission Set: Loan Approval Access (grant only to Branch Manager).
○ Use Permission Sets for temporary/extra privileges instead of changing base profiles.

8. Organization-Wide Defaults (OWD) & Sharing Rules

○ Set OWD to Private for sensitive objects (Customer/Loan).
○ Create Sharing Rules to give Branch Manager (or role) access to team records as required.

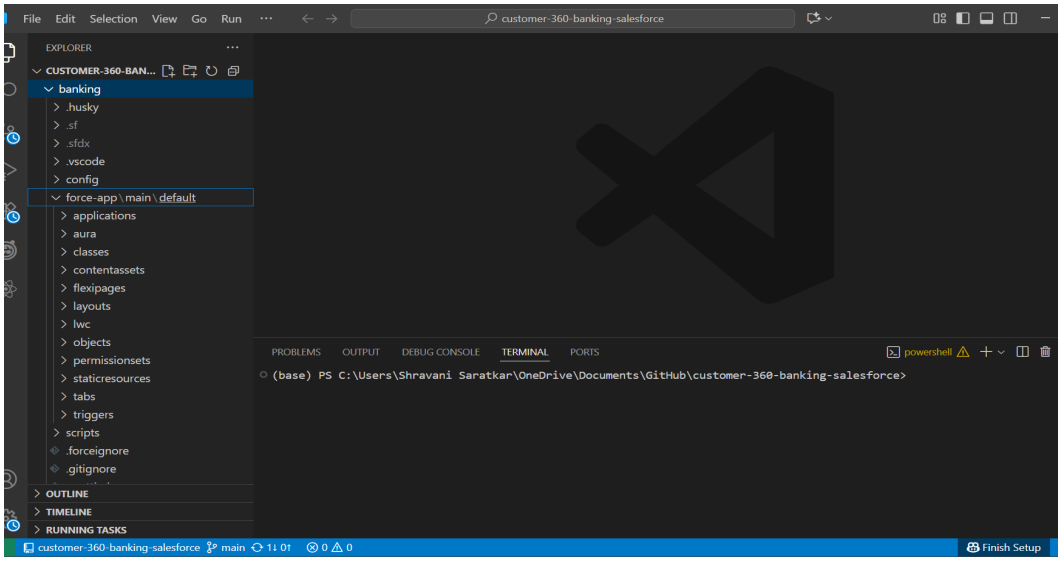
9. Login Access Policies

○ Configure Trusted IP Ranges and enable Admin “Login as User” for testing.
○ Optionally set session timeout and login hour restrictions.




10. Dev Org Setup & VS Code Authorization

○ Install Salesforce CLI and Salesforce Extensions in VS Code.
○ Authorize org: sfdx force:auth:web:login -banking (alias).
○ Confirm with sfdx force:org:list for metadata push/pull.



11. Sandbox Usage & Deployment Basics (notes)

    ○ For capstone: track metadata in a Git repo and use SFDX for deployment between orgs. We don't need  Sandbox in this project.

## Phase 3: Data Modelling And Relationships

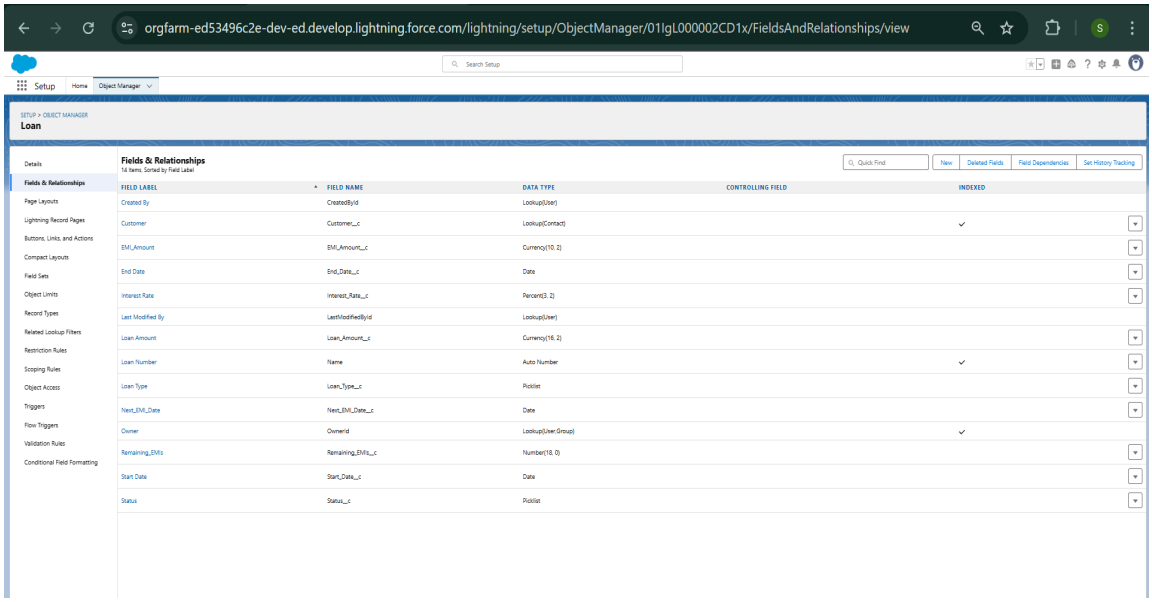
Goal: Design and implement object model capturing Customers, Loans, Transactions, and Financial Products.

1. Standard & Custom Objects

    ○ Standard: Account (corporate or bank entity), Contact (individual customer), Case (support).
    ○ Custom objects created:
        ■ Loan (Loan__c) — auto-number Loan Number (L-{0000}).
        ■ Transaction (Transaction__c) — auto-number Transaction ID (T-{0000}).
        ■ Financial Product (Financial_Product__c) — auto-number Product ID (FP-{0000}).

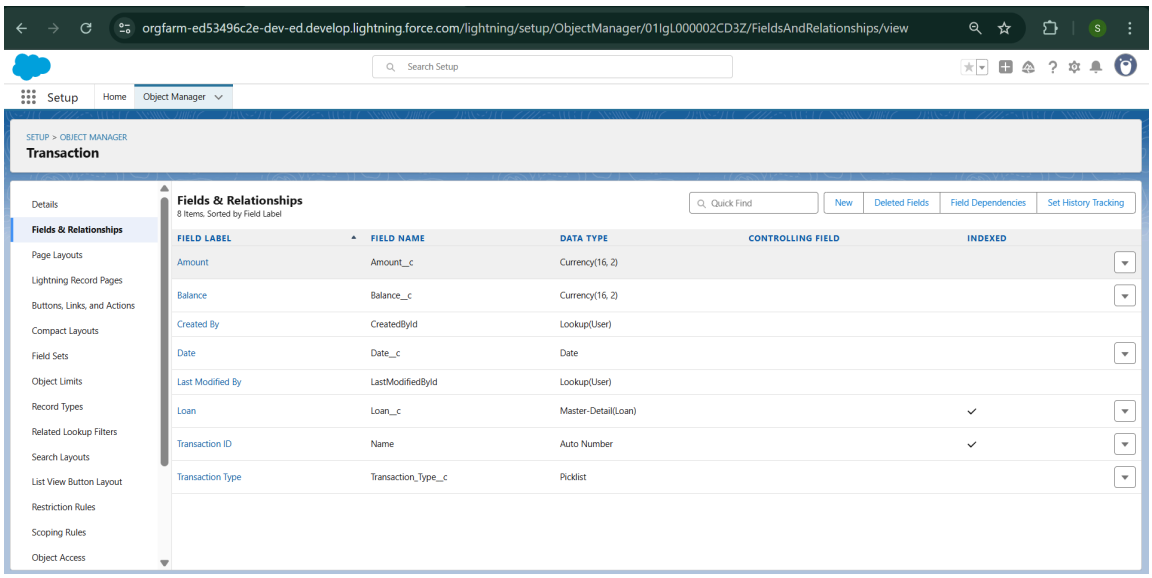
2. Fields (examples & types)

    ○ Loan__c:
    ■ Loan Amount — Currency (16,2)
    ■ Interest Rate — Percent (3,2)
    ■ Loan Type — Picklist (Home, Personal, Vehicle, Education)
    ■ Status — Picklist (Pending, Approved, Rejected, Closed)
    ■ Start Date — Date
    ■ End Date — Date
    ■ Customer — Lookup(Contact)



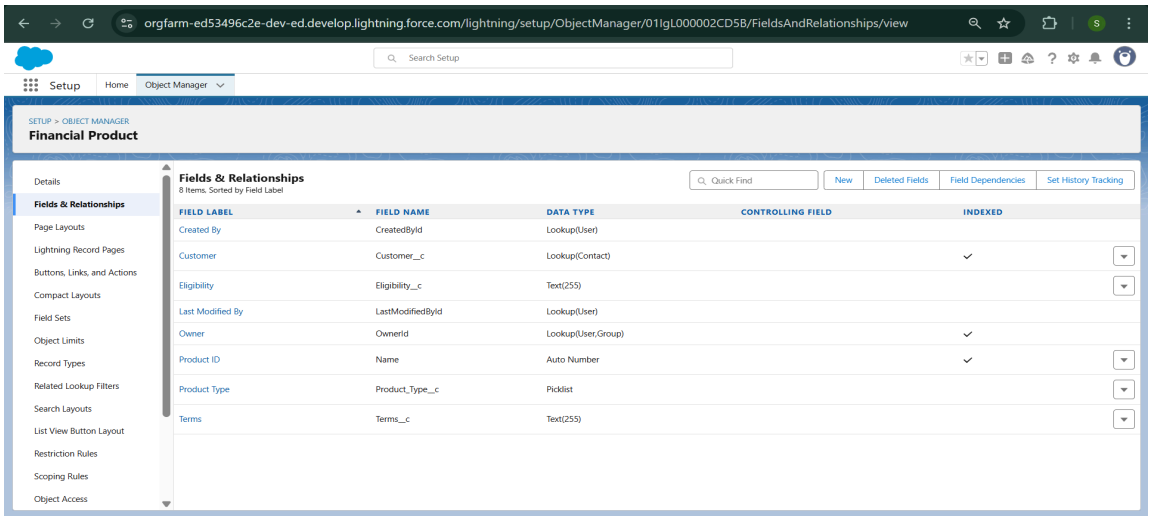

○ Transaction__c:

- Transaction Type — Picklist (Debit, Credit)
- Amount — Currency (16,2)
- Transaction Date — Date
- Balance — Currency (16,2)
- Related Loan — Master-Detail(Loan__c)




○ Financial_Product__c:

- Product Type — Picklist (Savings, Credit Card, Insurance, Investment)
- Terms — Text Area (255)
- Eligibility — Text Area (255)
- Customer — Lookup(Contact)




3. Relationships

   ○ Contact ↔ Loan — Lookup relationship (a contact can have many loans).
   ○ Loan ↔ Transaction — Master-Detail (loan is parent; transactions roll up to loan).
   ○ Contact ↔ Financial Product — Lookup (one-to-many).
   ○ (Optional) Junction Object if many-to-many is required (e.g., Customer_Product__c linking Contact and Financial_Product__c).

4. Record Types & Page Layouts

   ○ **Personal Loan** – standard fields like Loan Amount, Interest Rate, EMI, Start/End Date.

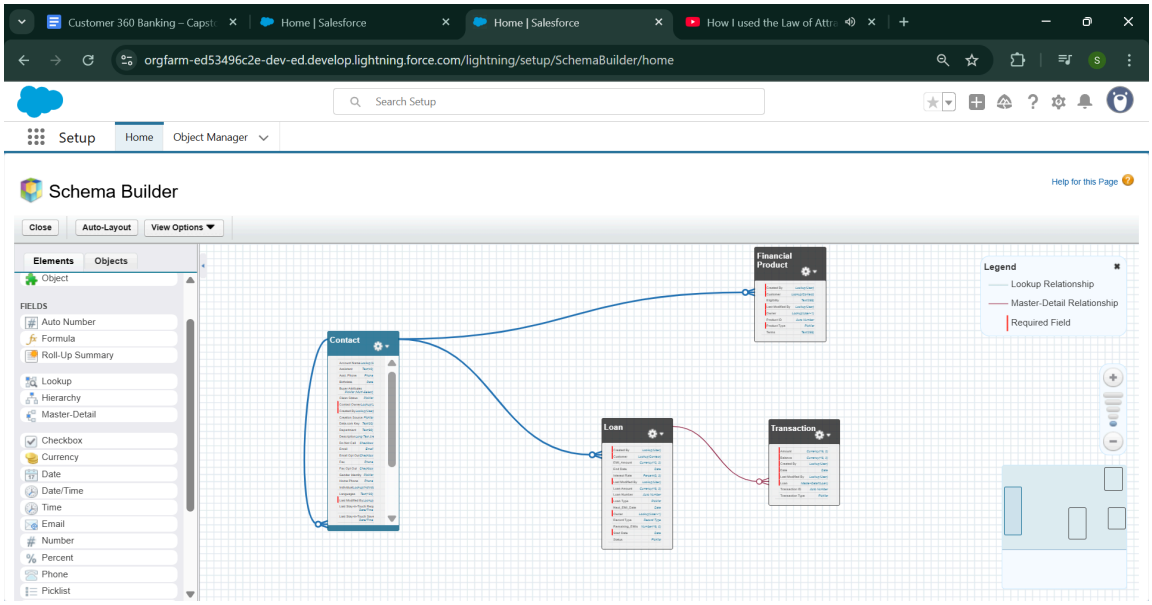
   **Business Loan** – in addition to loan basics, could include extra fields like Business Name, GSTIN (optional).
   Each record type is tied to its own **Page Layout**, so users see only the relevant fields.
   ○ Page Layouts: different layouts for Relationship Manager vs Branch Manager (Manager layout shows approval history & manager-only fields).

5. Compact Layouts

   ○ Configure compact layouts for Loan and Transaction so mobile/highlight panels show top fields: Loan Amount, Status, Next EMI Date / Transaction Date, Amount.

6. Schema Builder

   ○ Use Schema Builder to visualize object links and confirm relationships (Loan → Transactions, Contact → Loans, Contact → Products).




7. Lookup vs Master-Detail vs Hierarchical

   ○ Choose Lookup when records can exist independently (Contact → Loan).
   ○ Use Master-Detail when child should be deleted with parent and roll-up summaries are required (Transactions roll up to Loans).
   ○ Hierarchical relationships are used for linking users (not required here).

8. Junction Objects & External Objects

   ○ Junction Objects:Not required
   ○ External Objects: not required in this project

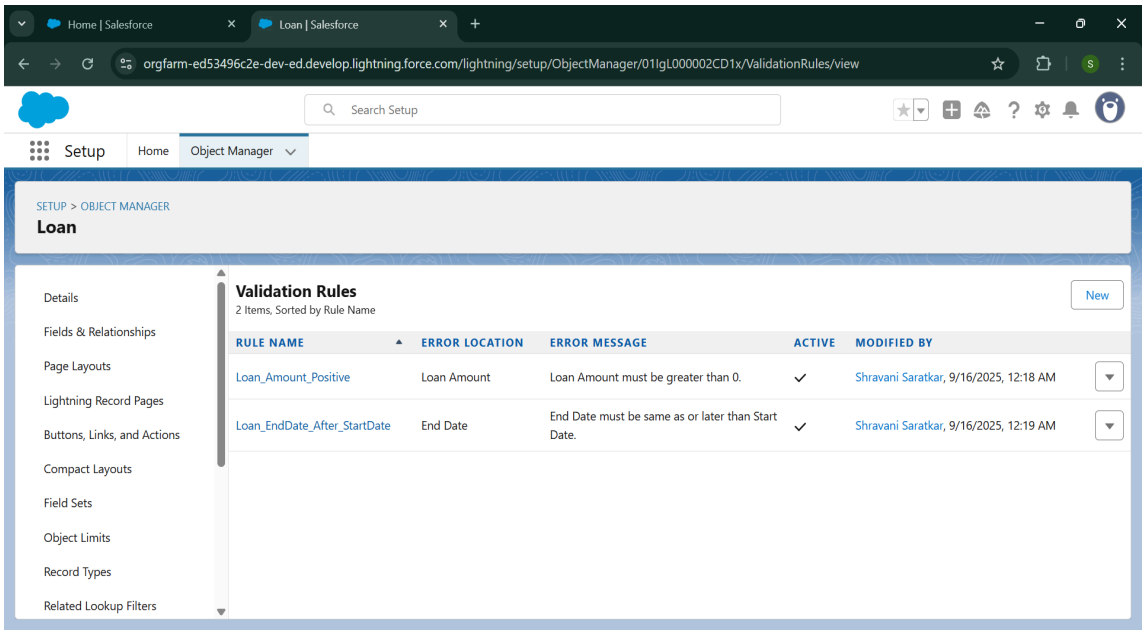
**Phase 3 Output / Next Steps**

○ Data model implemented in Salesforce with objects, fields, relationships, record types, and page layouts.
○ Ready to build Phase 4: Process Automation (Validation Rules, Flows, Approval Processes, Email Alerts) and Phase 5 Apex where complex logic (fraud detection, batch jobs) is required.

# Phase 4: Process Automation (Admin)

**Goal:** Automate loan application tasks and approvals.

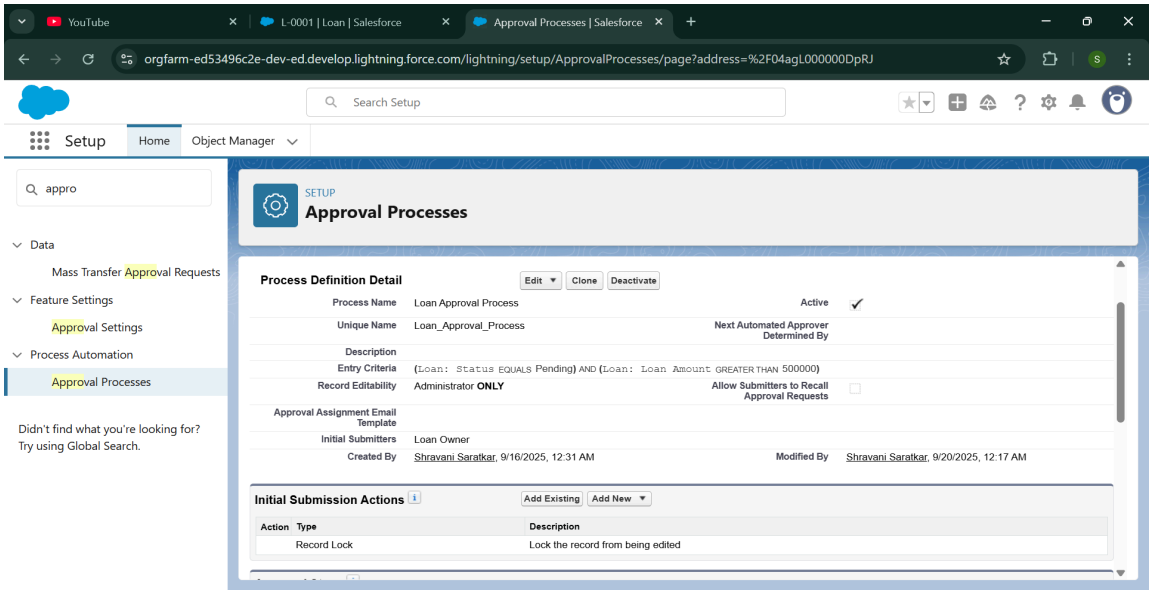
1. **Validation Rules**
   ○ Example: Loan End Date must be after Loan Start Date.
   ○ Example: Loan Amount must be greater than zero.



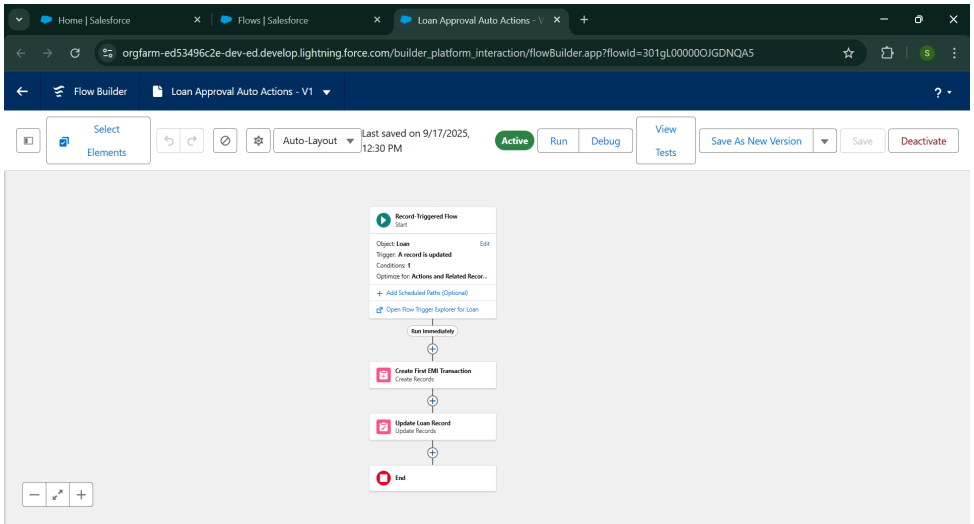

2. **Workflow Rules (legacy)**
   ○ Auto-send email when a new loan application is submitted (note: replaced by Flow in this project).

3. **Process Builder (legacy)**
   ○ Could auto-update Loan Status → replaced by Flow now.

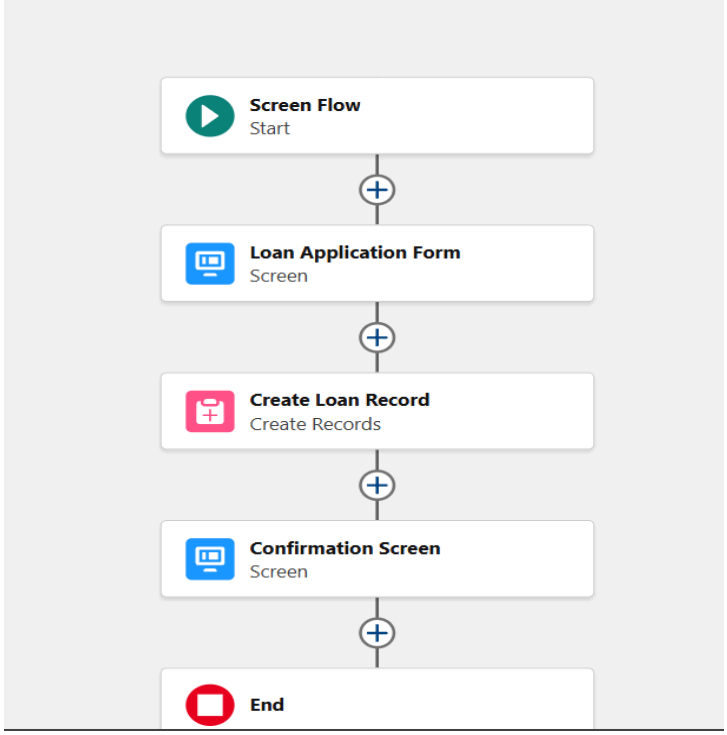
4. **Approval Process**
   ○ Loan > ₹5,00,000 → sent for Manager approval.
   ○ Manager can Approve or Reject the loan application.



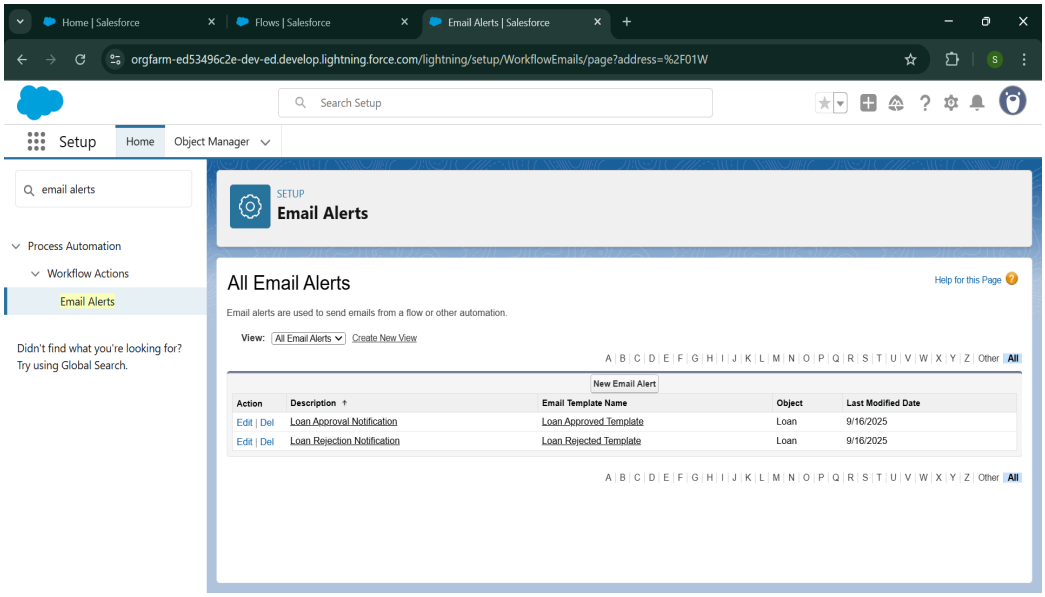

5. **Flow Builder**
   - **Record-Triggered Flow:** Automatically calculate EMI or Total Payable Amount when Loan Amount and Interest Rate are entered.



○ **Screen Flow:** Loan Application Form for customers to submit loan requests.




6. **Email Alerts**
   ○ Customer receives email notification after loan approval or rejection.




## Phase 4 Output / Next Steps

○ Validation Rules implemented to ensure data accuracy (e.g., Loan End Date > Start Date, Loan Amount > 0).

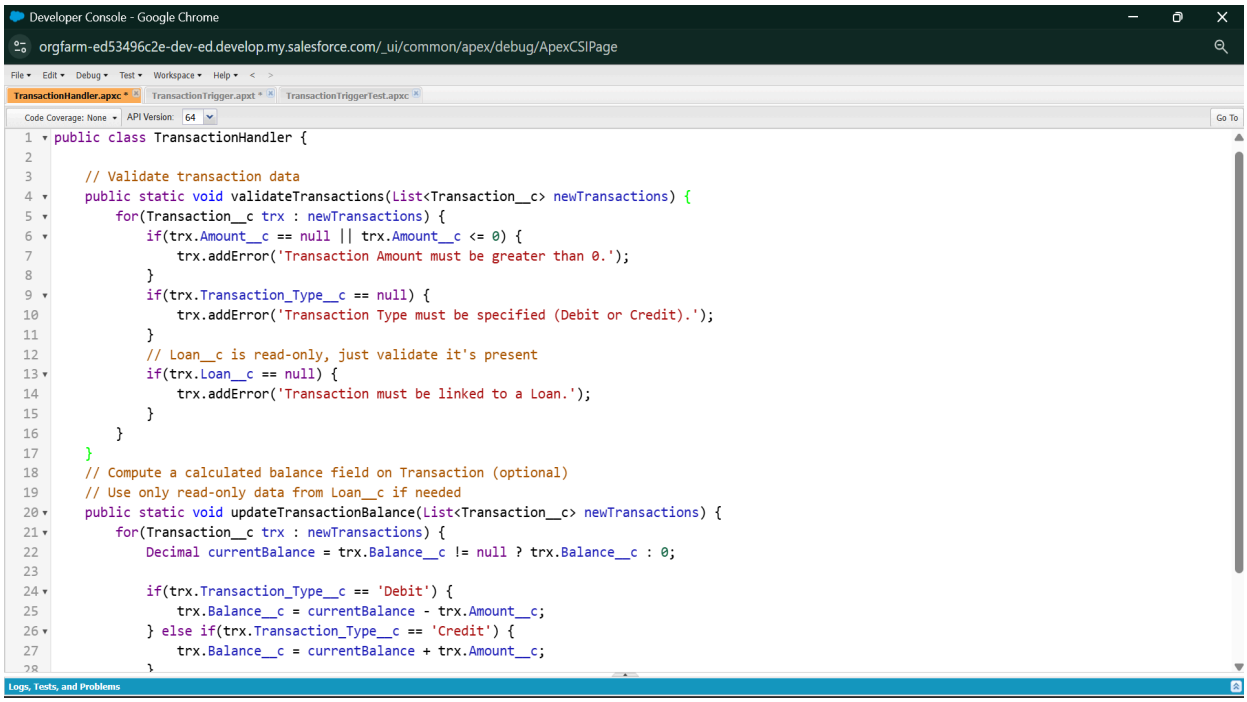
○ Approval Process configured for high-value loans (> ₹5,00,000) with Manager approval.
○ Flows implemented: Screen Flow for Loan Application Form and Record-Triggered Flow to calculate EMI / Total Payable Amount.
○ Email Alerts configured for Customers and Managers after submission and approval.
○ Ready to build **Phase 5: Apex Development**, where complex logic such as fraud detection, batch jobs, and advanced automations will be implemented.

# Phase 5: Apex Programming (Developer)

**Goal:** Add advanced logic for Transactions and Loans using Apex.

## 1. Classes & Objects

- `TransactionHandler` class created for reusable logic.
- Handles validation of transaction data and balance calculations.



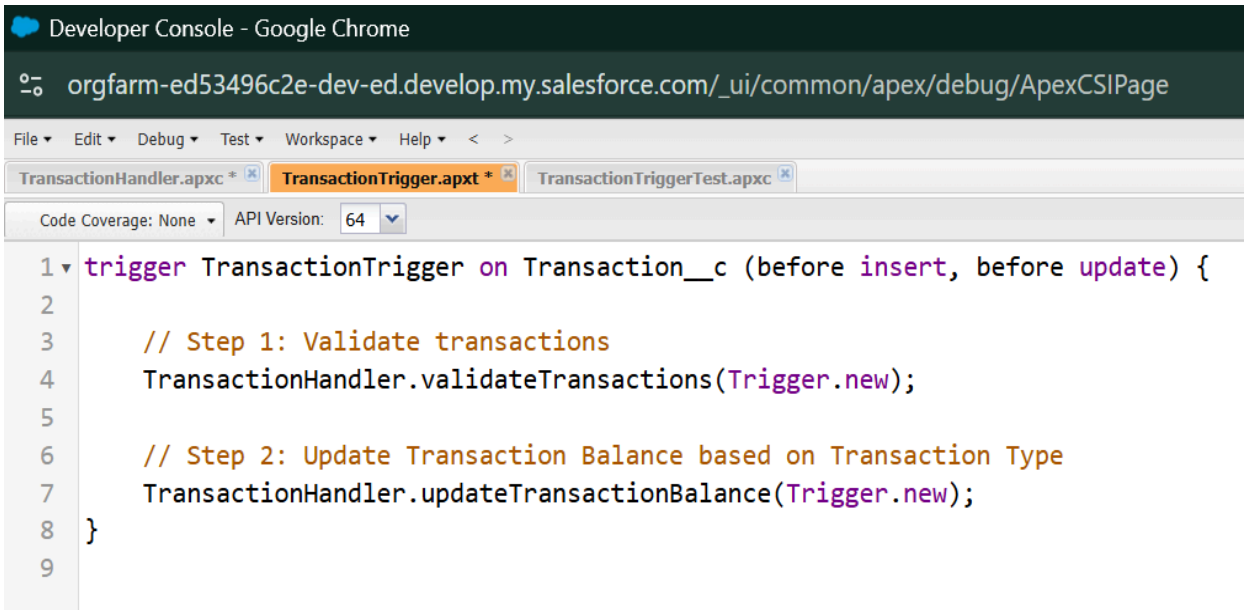

```
public class TransactionHandler {

    // Validate transaction data
    public static void validateTransactions(List<Transaction__c> newTransactions) {
        for(Transaction__c trx : newTransactions) {
            if(trx.Amount__c == null || trx.Amount__c <= 0) {
                trx.addError('Transaction Amount must be greater than 0.');
            }
            if(trx.Transaction_Type__c == null) {
                trx.addError('Transaction Type must be specified (Debit or Credit).');
            }
            // Loan__c is read-only, just validate it's present
            if(trx.Loan__c == null) {
                trx.addError('Transaction must be linked to a Loan.');
            }
        }
    }
    // Compute a calculated balance field on Transaction (optional)
    // Use only read-only data from Loan__c if needed
    public static void updateTransactionBalance(List<Transaction__c> newTransactions) {
        for(Transaction__c trx : newTransactions) {
            Decimal currentBalance = trx.Balance__c != null ? trx.Balance__c : 0;

            if(trx.Transaction_Type__c == 'Debit') {
                trx.Balance__c = currentBalance - trx.Amount__c;
            } else if(trx.Transaction_Type__c == 'Credit') {
                trx.Balance__c = currentBalance + trx.Amount__c;
            }
```

## 2. Apex Triggers

- `TransactionTrigger` implemented on **Transaction__c** (before insert/update).

- Ensures:
    - Amount > 0
    - Transaction Type is specified
    - Linked Loan exists

- Updates `Balance__c` for each transaction based on Credit/Debit.




### 3. Trigger Design Pattern

- Logic separated into **TransactionHandler** class.
- Trigger only calls handler methods → modular and maintainable.

### 4. SOQL & SOSL

- Trigger queries **Loan__c** when needed to validate linked loan records.
- Bulk-safe queries implemented to handle multiple transactions efficiently.

### 5. Collections: List, Set, Map

- Lists and Maps used to process multiple transactions and handle bulk operations.
- Ensures that bulk inserts/updates are processed correctly.

### 6. Control Statements

- IF statements used to validate transactions:
  - Amount ≤ 0 → throws error
  - Missing Transaction Type → throws error
  - Missing Loan → throws error

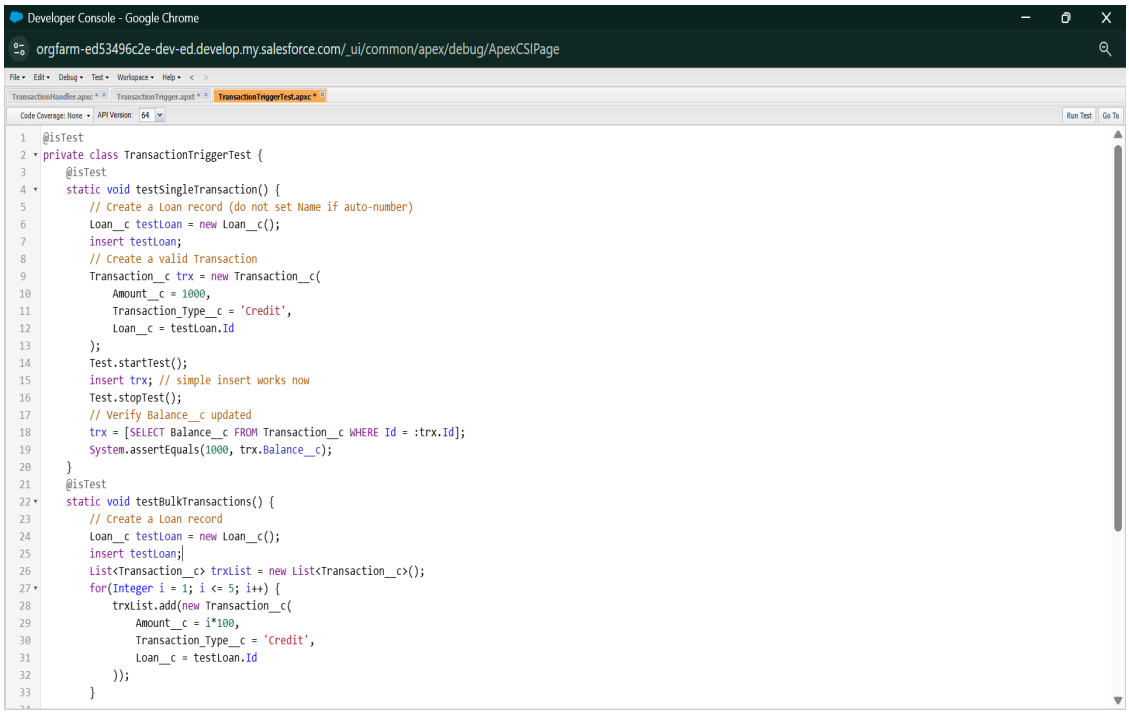
### 7. Exception Handling

- Errors during validation are caught and displayed to the user.

- Prevents invalid transactions from being saved.

## 8. Test Classes

- `TransactionTriggerTest` created to cover:
  - Single Credit and Debit transactions
  - Bulk transactions
  - Invalid data scenarios (negative amount, missing type, missing Loan)
- Ensures 100% code coverage and robust validation.




```
@isTest
private class TransactionTriggerTest {
    @isTest
    static void testSingleTransaction() {
        // Create a Loan record (do not set Name if auto-number)
        Loan__c testLoan = new Loan__c();
        insert testLoan;
        // Create a valid Transaction
        Transaction__c trx = new Transaction__c(
            Amount__c = 1000,
            Transaction_Type__c = 'Credit',
            Loan__c = testLoan.Id
        );
        Test.startTest();
        insert trx; // simple insert works now
        Test.stopTest();
        // Verify Balance__c updated
        trx = [SELECT Balance__c FROM Transaction__c WHERE Id = :trx.Id];
        System.assertEquals(1000, trx.Balance__c);
    }
    @isTest
    static void testBulkTransactions() {
        // Create a Loan record
        Loan__c testLoan = new Loan__c();
        insert testLoan;
        List<Transaction__c> trxList = new List<Transaction__c>();
        for(Integer i = 1; i <= 5; i++) {
            trxList.add(new Transaction__c(
                Amount__c = i*100,
                Transaction_Type__c = 'Credit',
                Loan__c = testLoan.Id
            ));
        }
```

## Phase 5 Output / Next Steps

- TransactionHandler Apex class implemented with validation and balance calculation logic.
- TransactionTrigger implemented (bulk-safe, modular).
- Test class implemented and all tests pass.
- Trigger functionality verified for single and bulk transactions.
- Ready to proceed for phase 6.