Phase 1: Problem Understanding & Industry Analysis

Requirement Gathering

- Identify gaps in manual payment tracking.
- Understand issues in sending timely payment reminders.

Stakeholder Analysis

- Customers, bank staff, recovery agents, and managers.
- Focus on their roles and interaction with the payment system.

Business Process Mapping

- Map repayment lifecycle:

  due date → reminder → payment → overdue → recovery.
- Visualize processes for loan and EMI repayments.

Industry-specific Use Case Analysis

- EMI reminders, loan repayments, credit card dues monitoring.
- Streamline overdue and recovery workflows.

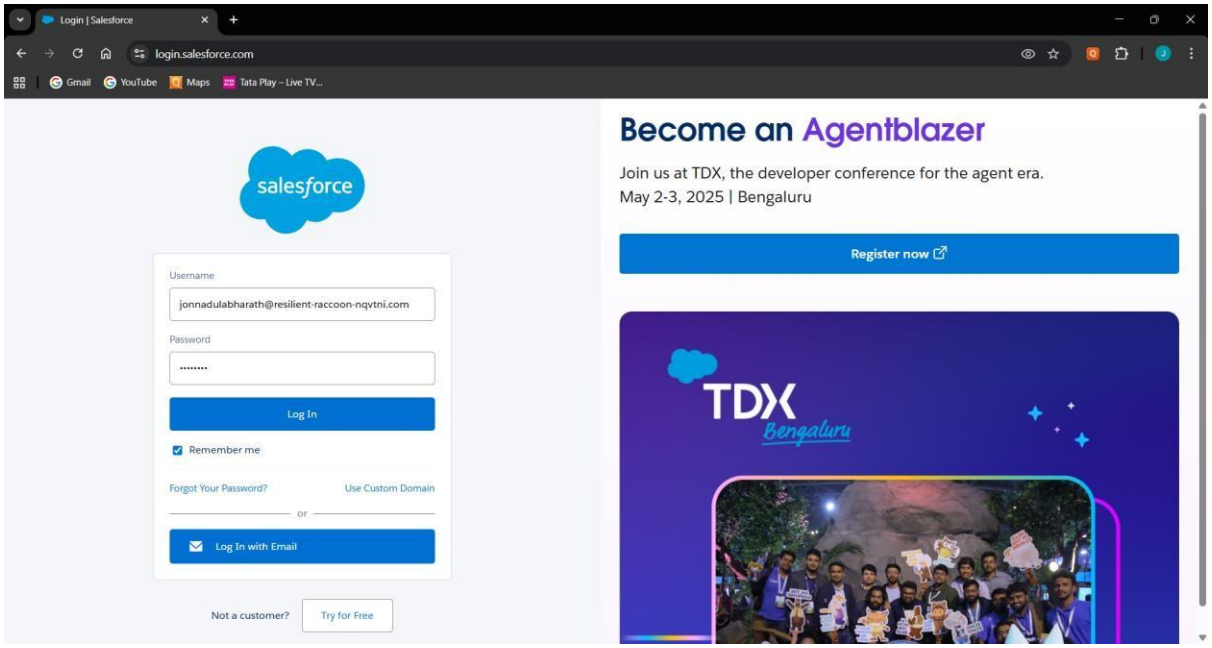
AppExchange Exploration

- Research payment reminder.
- Identify ready-made apps to speed up implementation.

Phase 2: Org Setup & Configuration

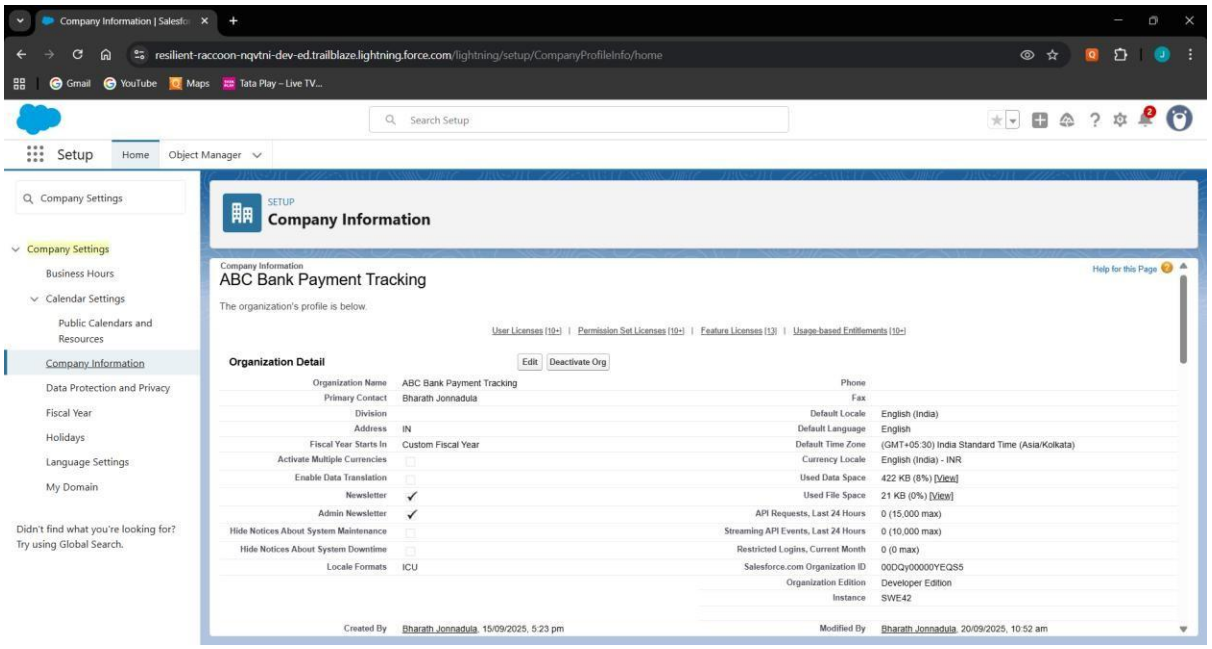
Salesforce Editions

- Use Developer Edition for initial development and testing.
- Plan for Enterprise Edition for production banking workflows.



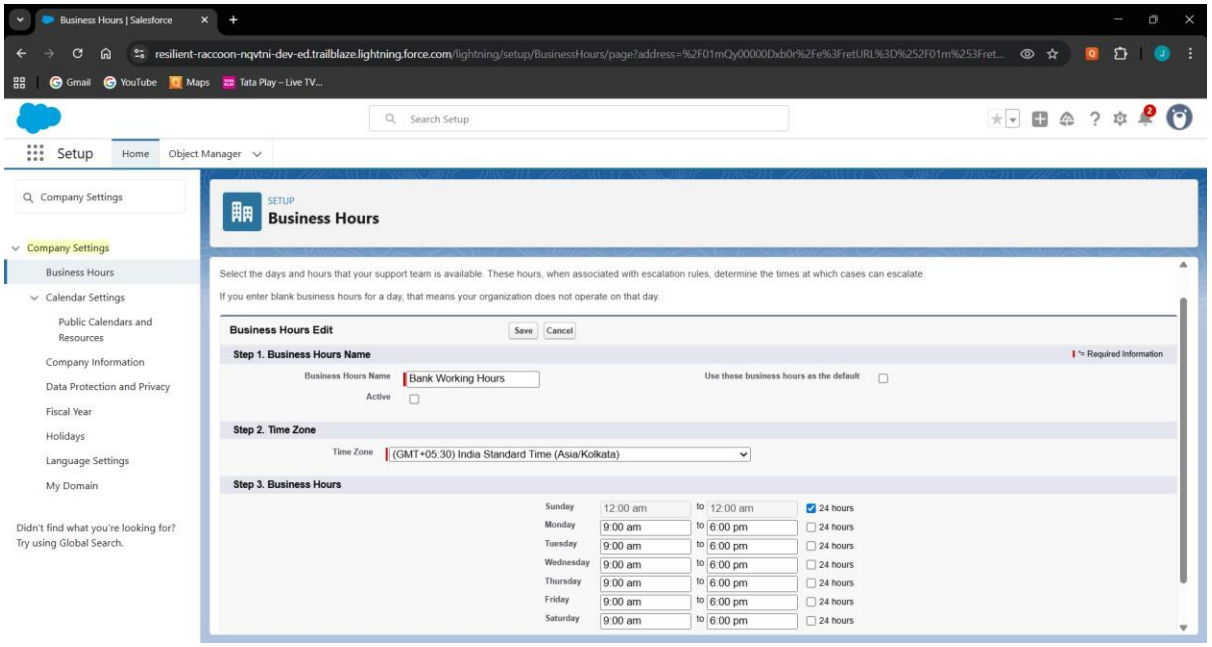
Company Profile Setup

- Set Company Name, Currency (INR/USD), and Time Zone.
- Configure organization-wide defaults.



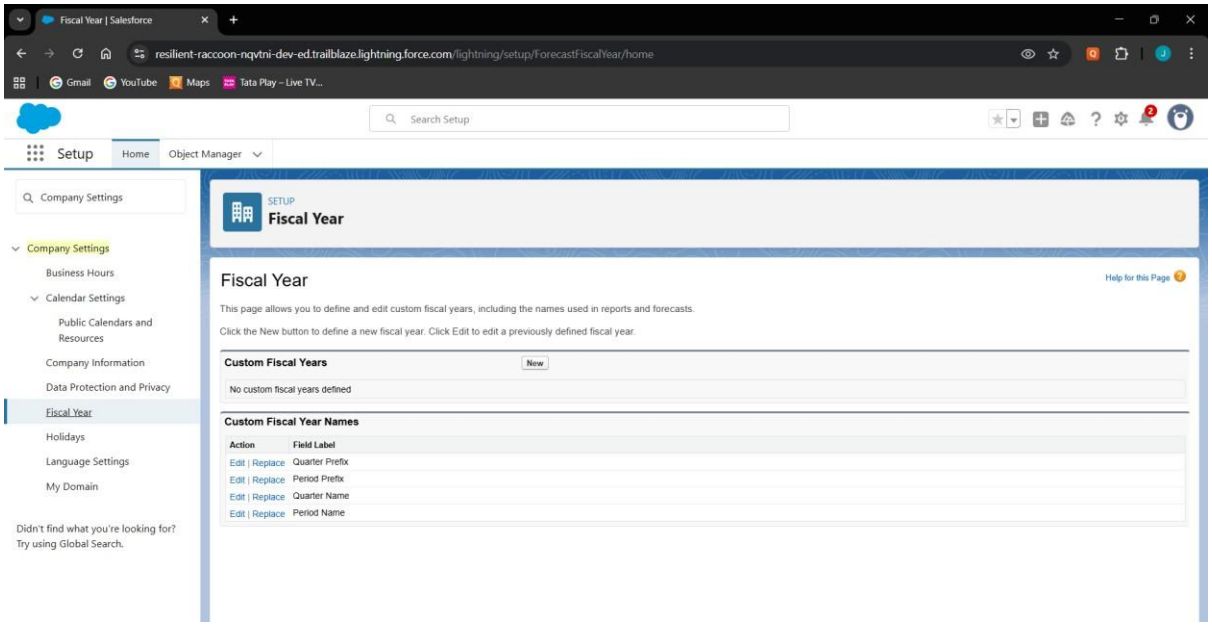
Business Hours & Holidays

- Define working hours (e.g., Mon–Sat, 9 AM–6 PM).
- Add holidays like Independence Day and Diwali.



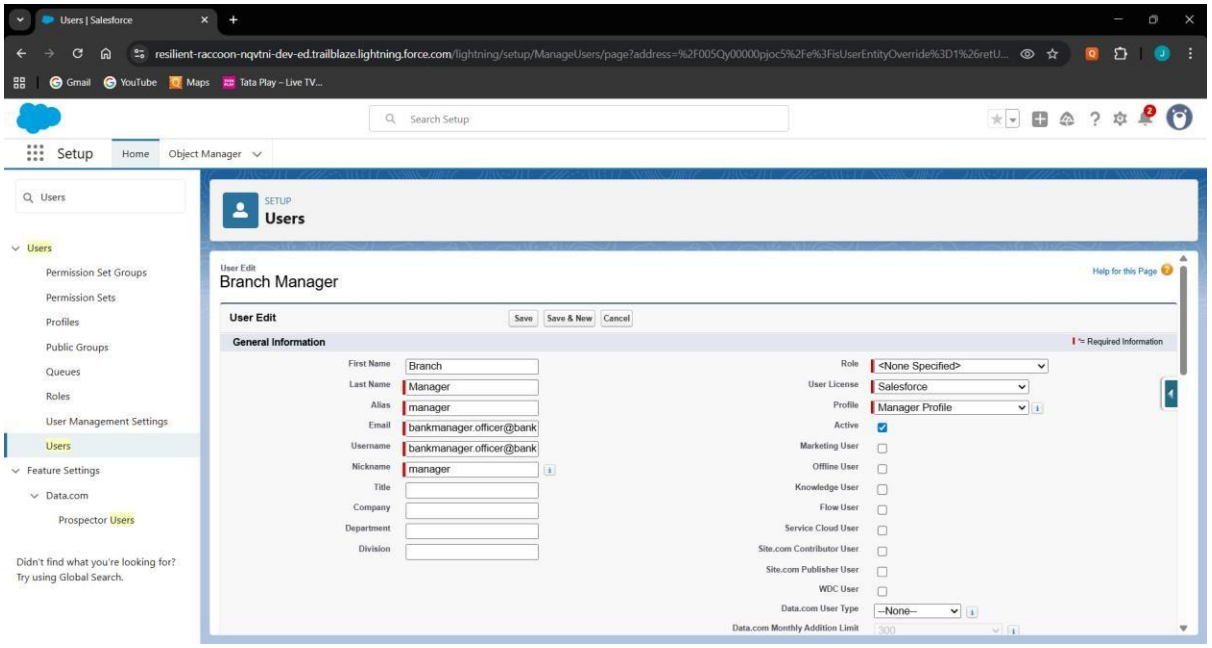
Fiscal Year Settings

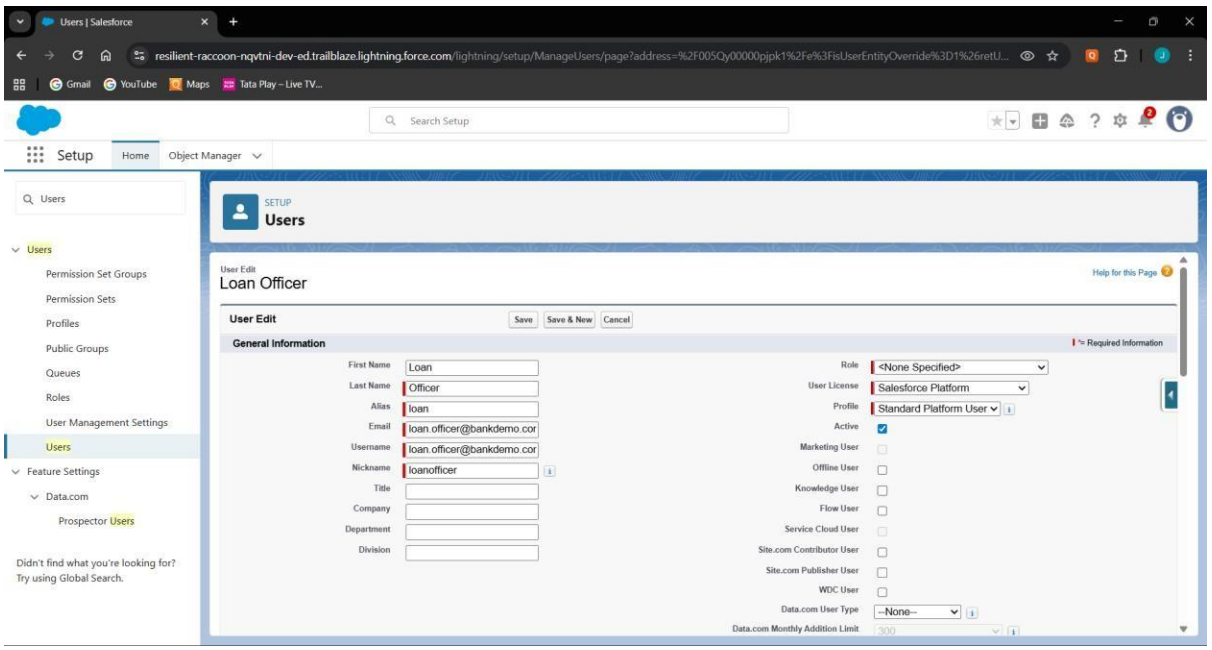
- Use standard Jan–Dec fiscal year or custom if needed.
- Align with financial reporting requirements.

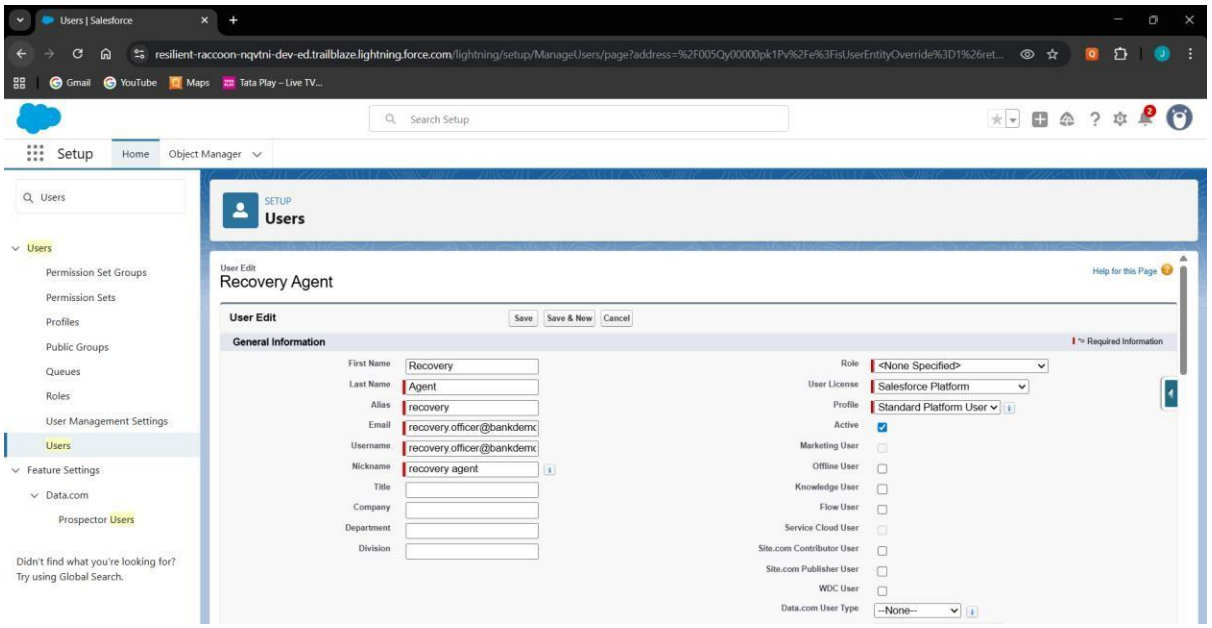


User Setup & Licenses

- Create Loan Officer, Recovery Agent, and Manager users.
- Assign appropriate licenses (Salesforce / Salesforce Platform).

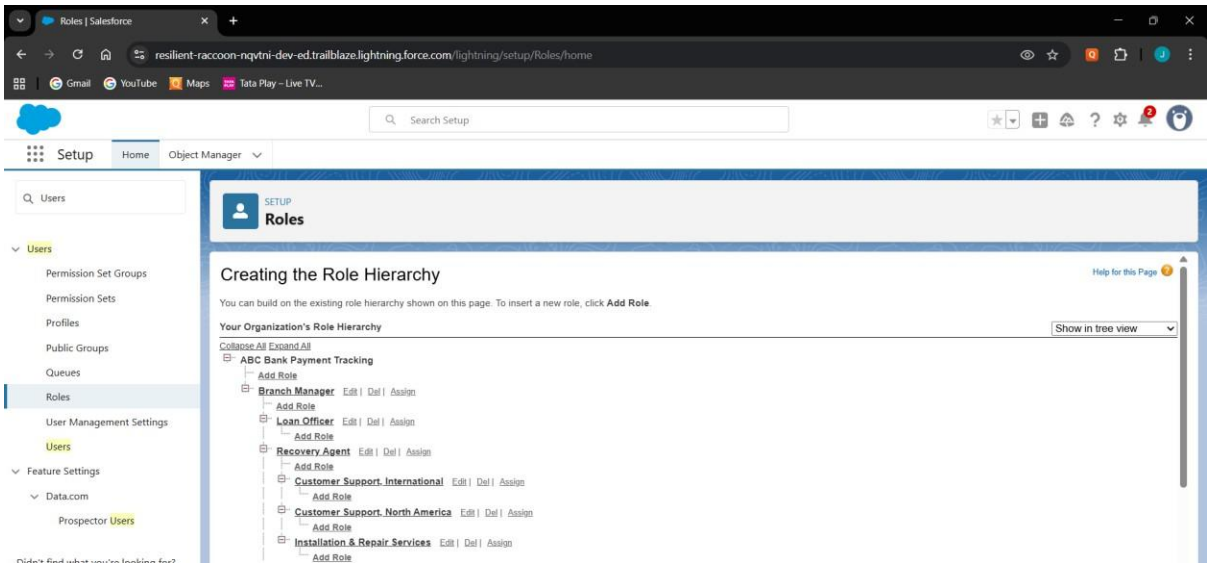






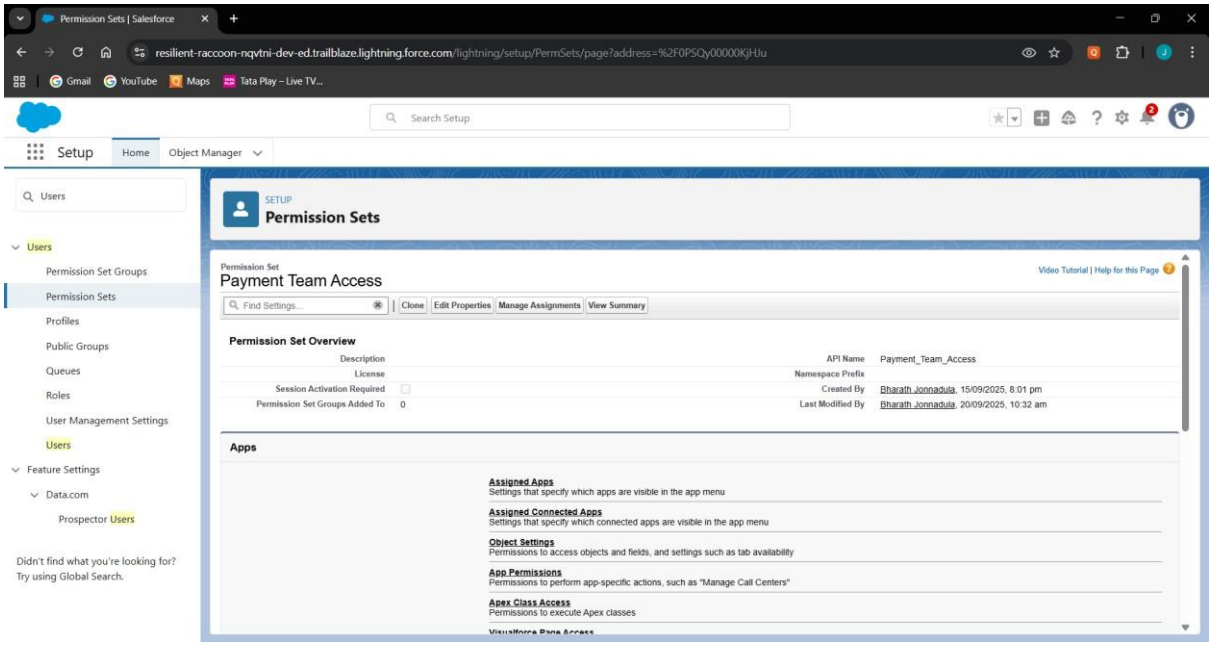
Profiles & Roles

- Create custom profiles (Loan Officer, Manager, Recovery Agent).
- Establish role hierarchy (Manager → Loan Officer / Recovery Agent).



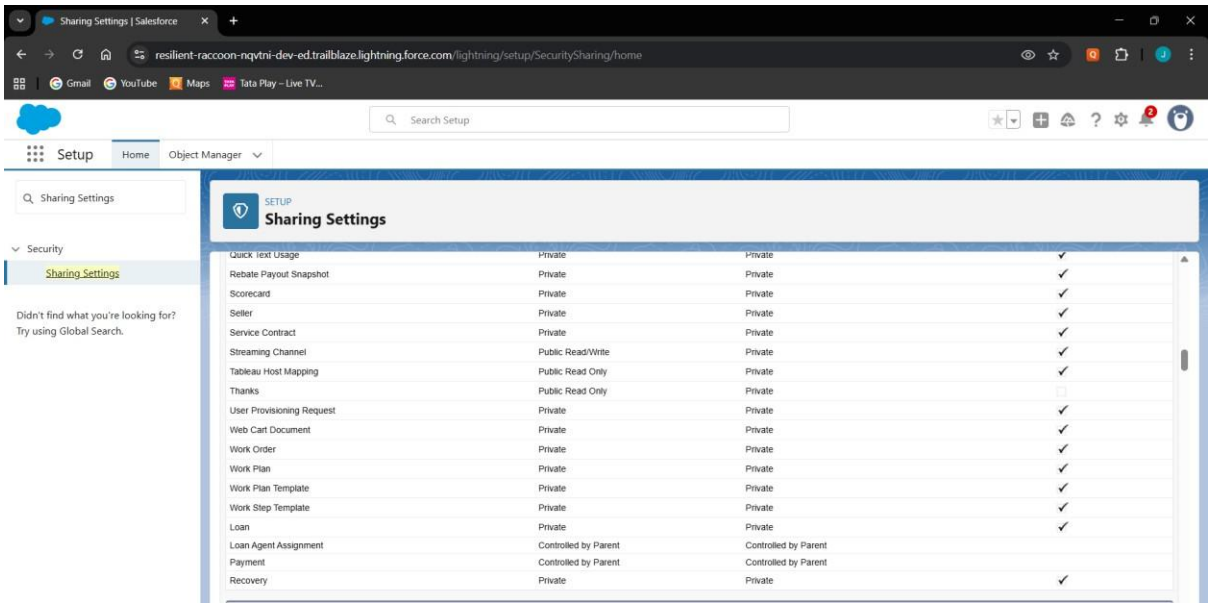
Permission Sets

- Create a Permission Set for payment tracking access.
- Assign Permission Set to Loan Officers and Managers.



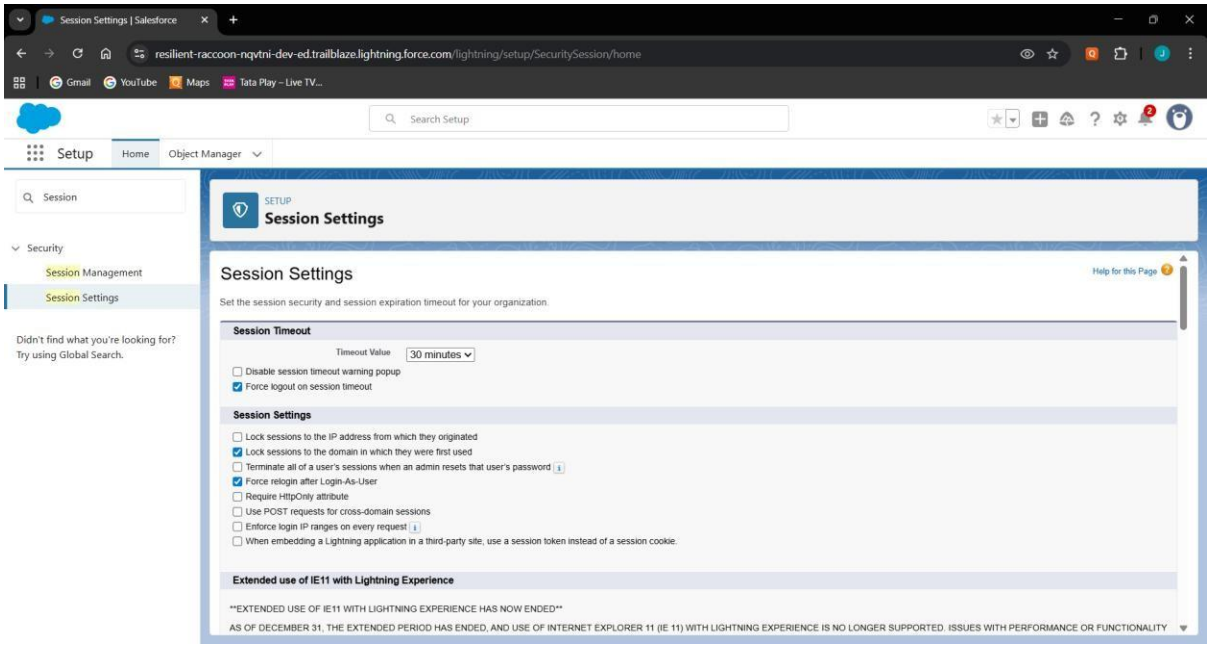
OWD & Sharing Rules

- Set custom objects (Loan, Payment, Recovery) to Private.
- Create rules to share overdue records with Recovery Agents.



Login Access Policies

- Set session timeout to 30 minutes.
- Enable login hours (8 AM–8 PM) for users.

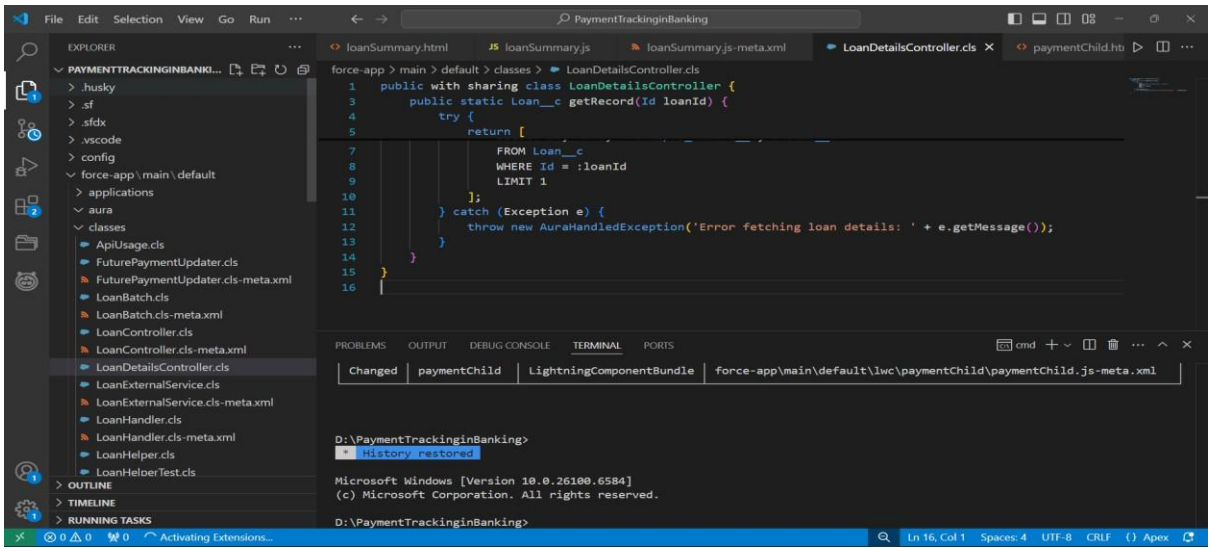


Dev Org & Sandbox Usage

- Use Developer Org for development and testing.
- Plan sandbox usage for production-like testing later.

Deployment Basics

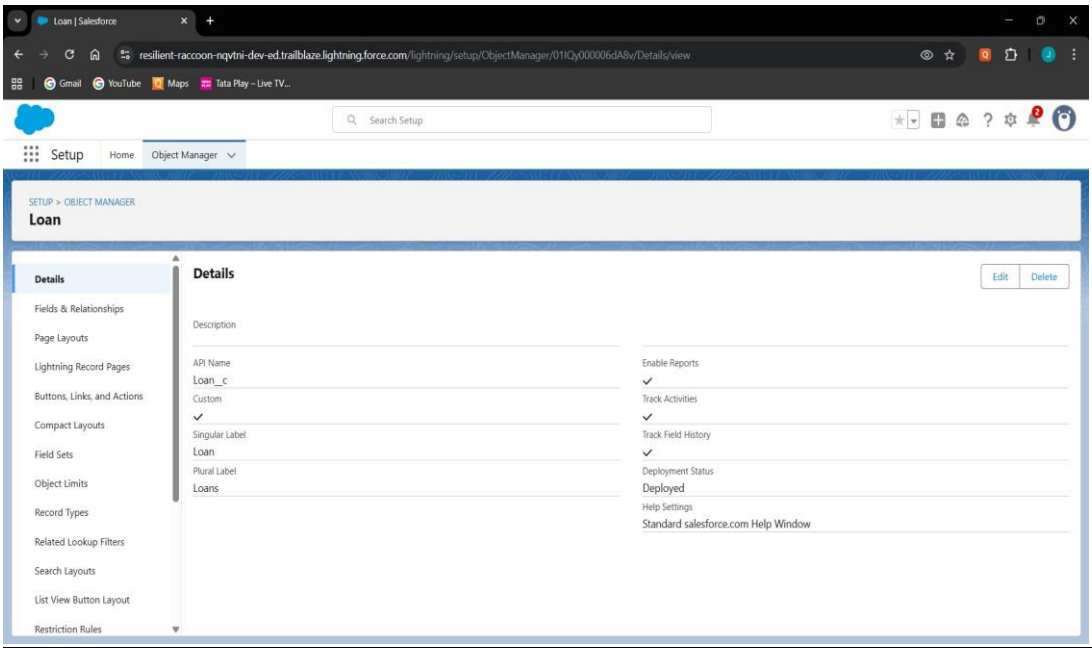
- Use SFDX commands to push/pull metadata.
- Prepare for deployment via Change Sets or SFDX when ready.

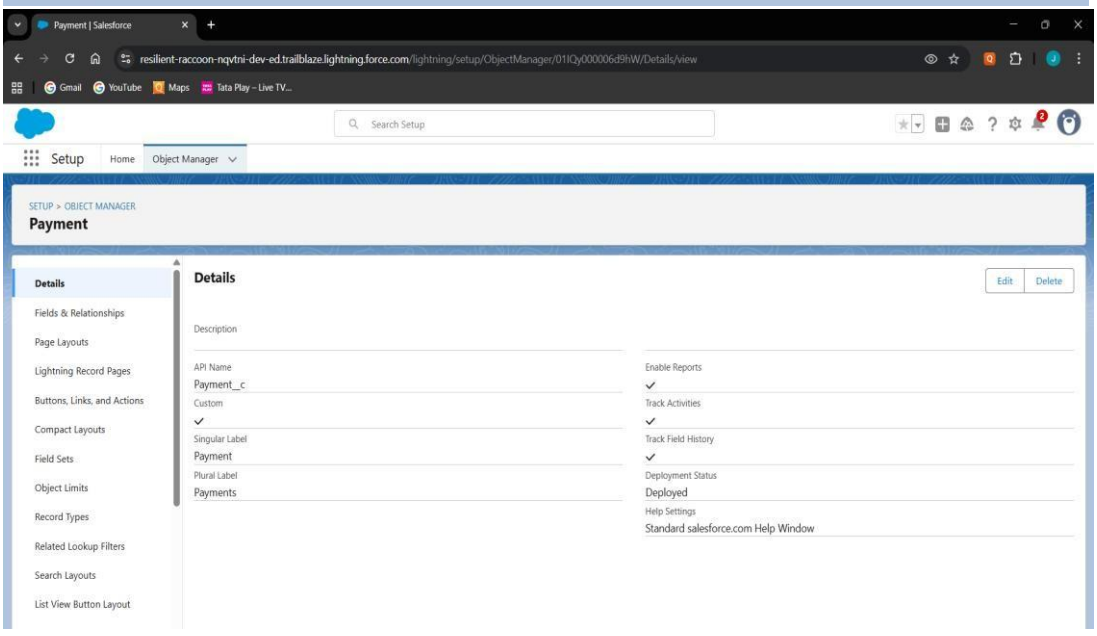


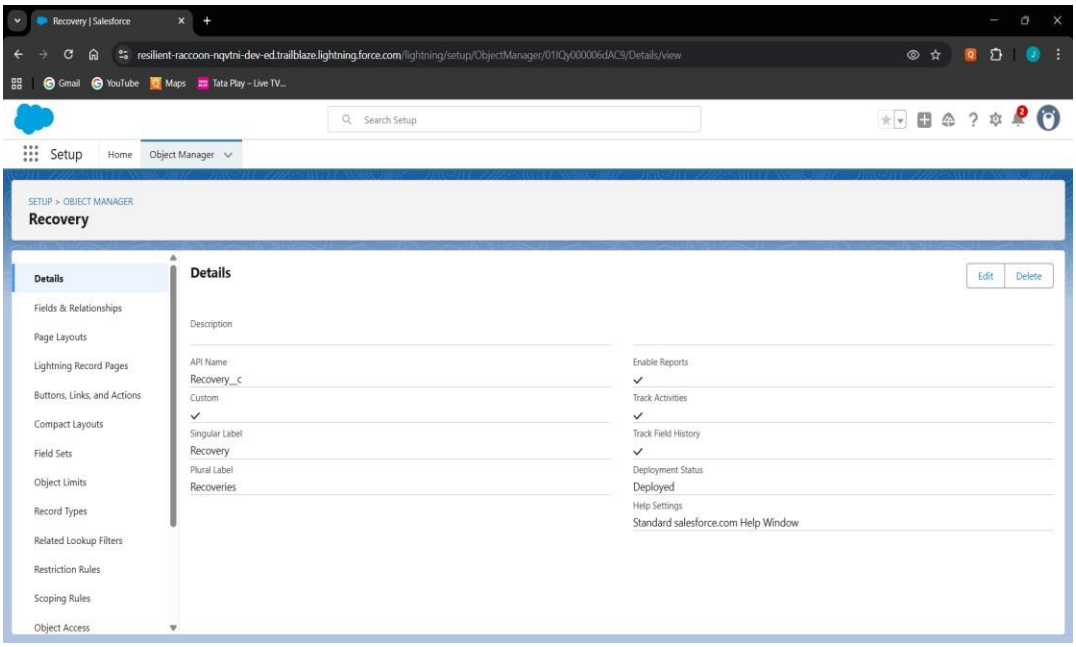
Phase 3: Data Modeling & Relationships

Standard & Custom Objects

- Use Account/Contact for Customers.
- Create custom objects: Loan, Payment, Recovery, Loan Agent Assignment (junction).

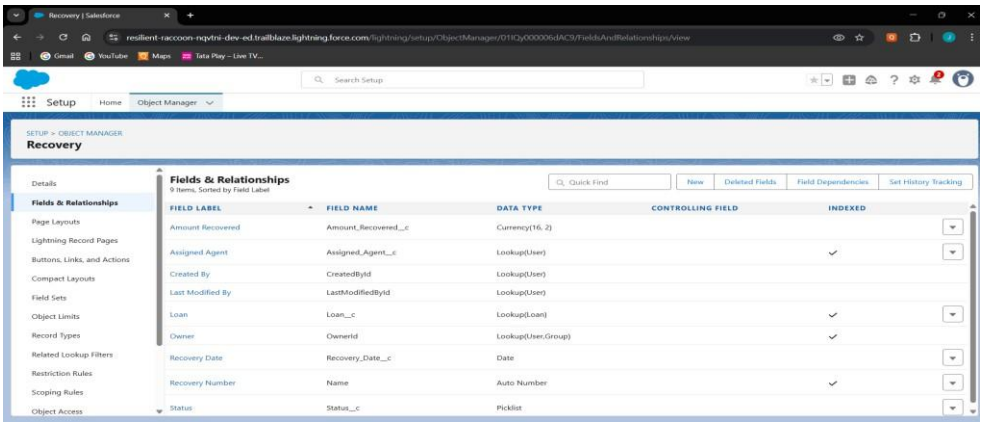






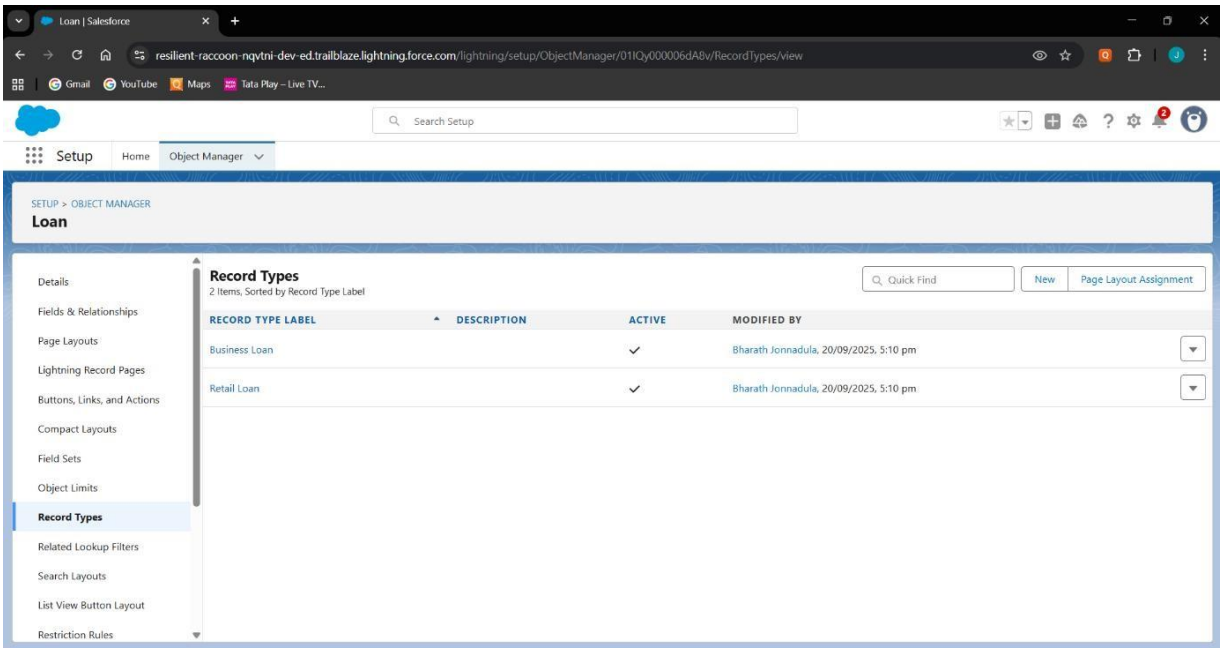
Fields

- Loan: Loan Number (Auto Number), Principal Amount (Currency), Interest Rate (Percent), Term Months (Number), Start Date (Date), Status (Picklist), Customer (Lookup to Account/Contact).
- Payment: Payment Date (Date), Amount (Currency), Payment Method (Picklist: Cash/UPI/Transfer), Status (Picklist: Paid/Overdue), Loan (Master-Detail).
- Recovery: Recovery Date (Date), Amount Recovered (Currency), Status (Picklist: Assigned/InProgress/Completed), Loan (Lookup), Assigned Agent (Lookup to User).
- Loan Agent Assignment (junction): Loan (Master-Detail), Assigned Agent (Lookup to User).



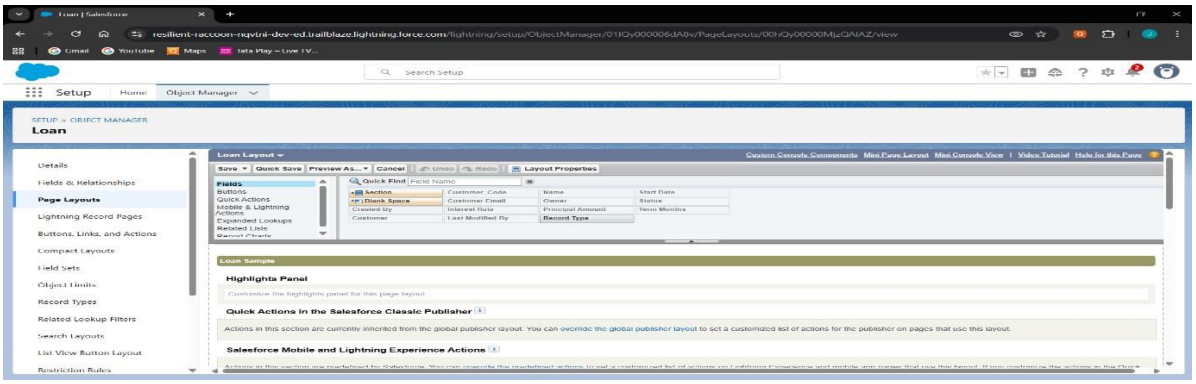
Record Types

- Loan: Retail Loan & Business Loan.
- Configure Status picklist values per record type (e.g., Retail = Active/Closed, Business = Active/Defaulted).



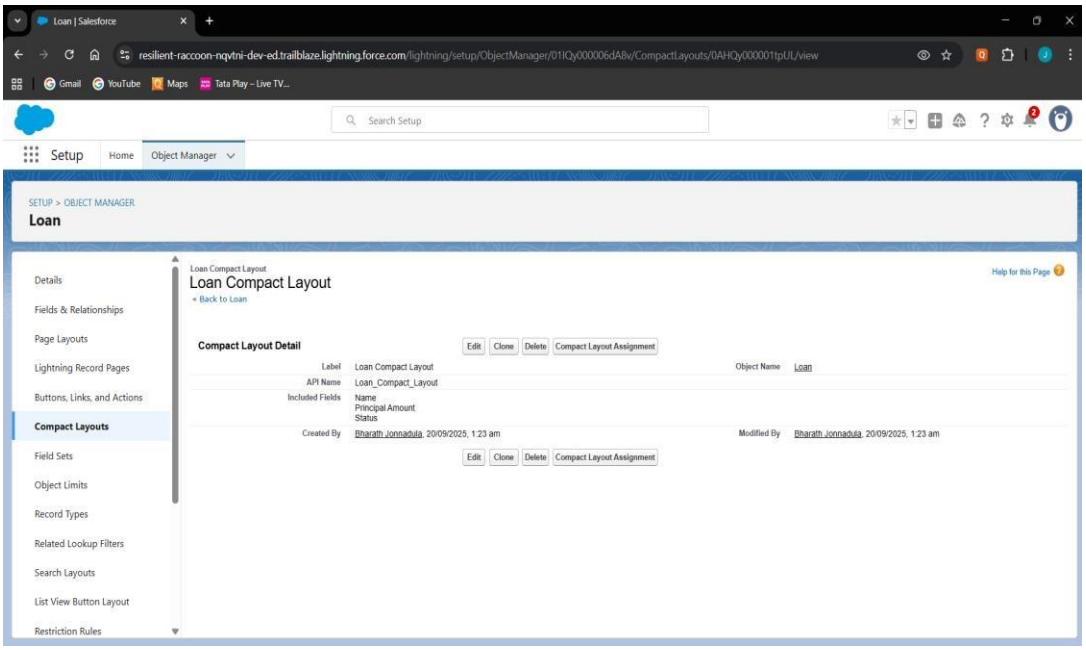
Page Layouts

- Loan Layout: sections for Basic Info, Financials, Dates & Status. Related Lists = Payments, Recoveries, Loan Agent Assignments.
- Payment Layout: fields for Date, Amount, Method, Status.
- Recovery Layout: fields for Date, Amount Recovered, Status, Assigned Agent.



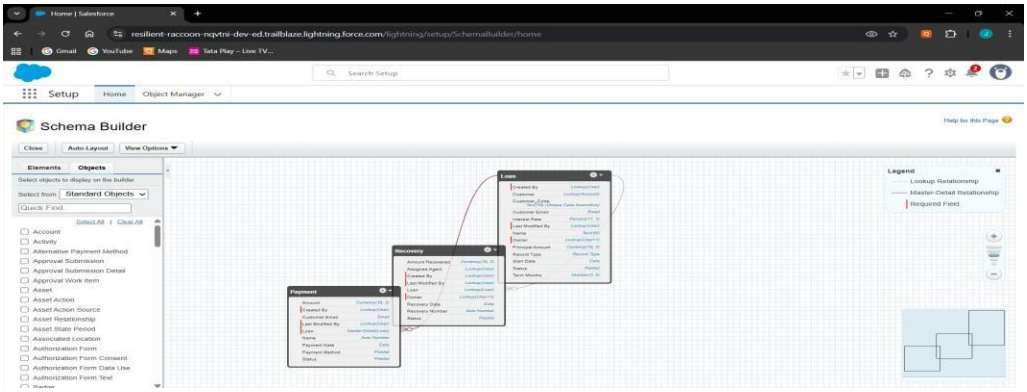
Compact Layouts

- Loan Compact Layout: Loan Number, Principal Amount, Status, Customer.
- Payment Compact Layout: Payment Date, Amount, Status.
- Recovery Compact Layout: Recovery Date, Amount Recovered, Status.



Schema Builder

- Visualize Loan, Payment, Recovery, and Loan Agent Assignment objects.
- Confirm relationships: Loan–Payment (Master-Detail), Loan–Recovery (Lookup), Loan–LoanAgentAssignment (Master-Detail), LoanAgentAssignment–User (Lookup).



Lookup vs Master-Detail vs Hierarchical

- Payment → Loan = Master-Detail (payments always tied to loans).
- Recovery → Loan = Lookup (independent audit records).
- Loan Agent Assignment → Loan = Master-Detail, → User = Lookup.
- Hierarchical not used (only for User object).

Junction Objects

- Loan Agent Assignment to handle many-to-many between Loans and Agents.
- Enables multiple agents handling multiple loans.

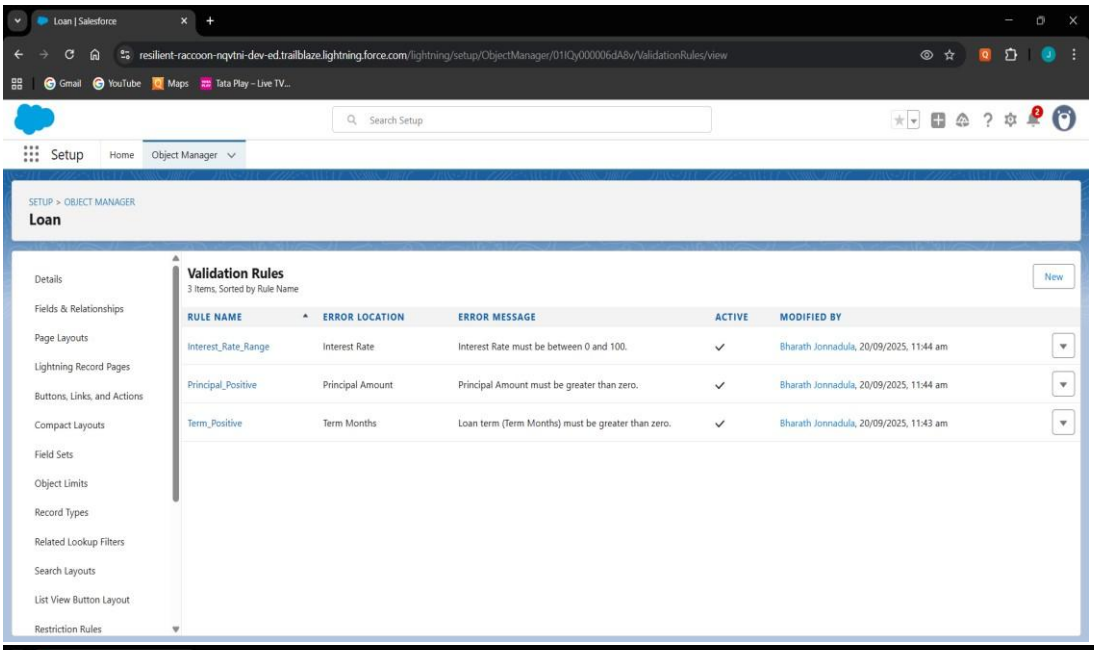
External Objects

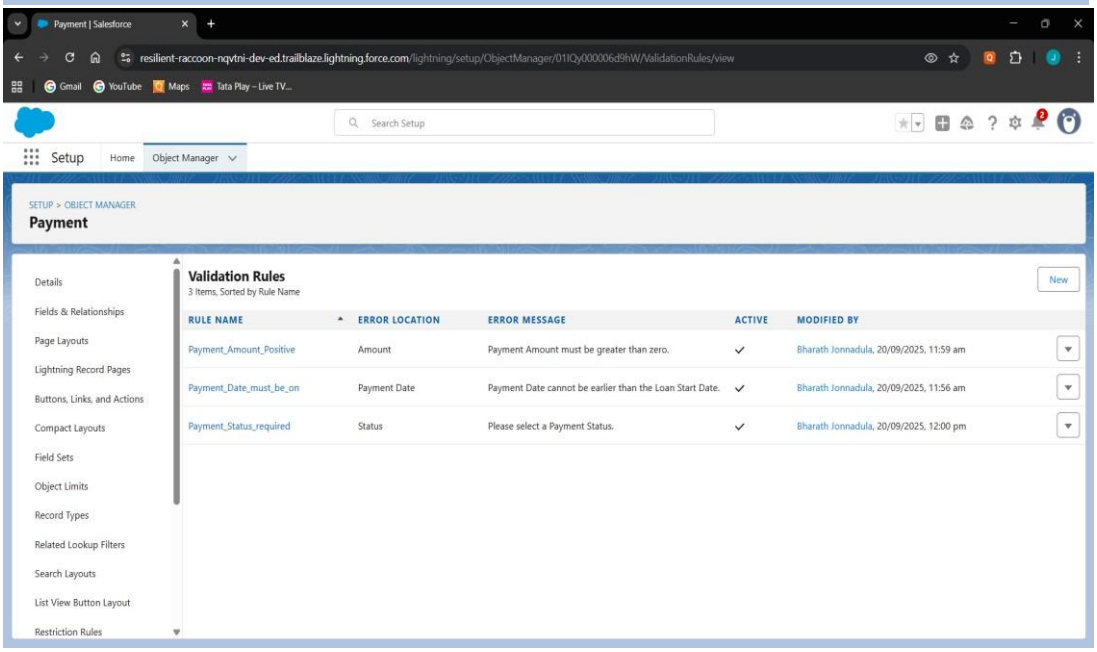
- Concept: Connect to external payment system using Salesforce Connect.
- Example: Show legacy payment history without storing inside Salesforce.

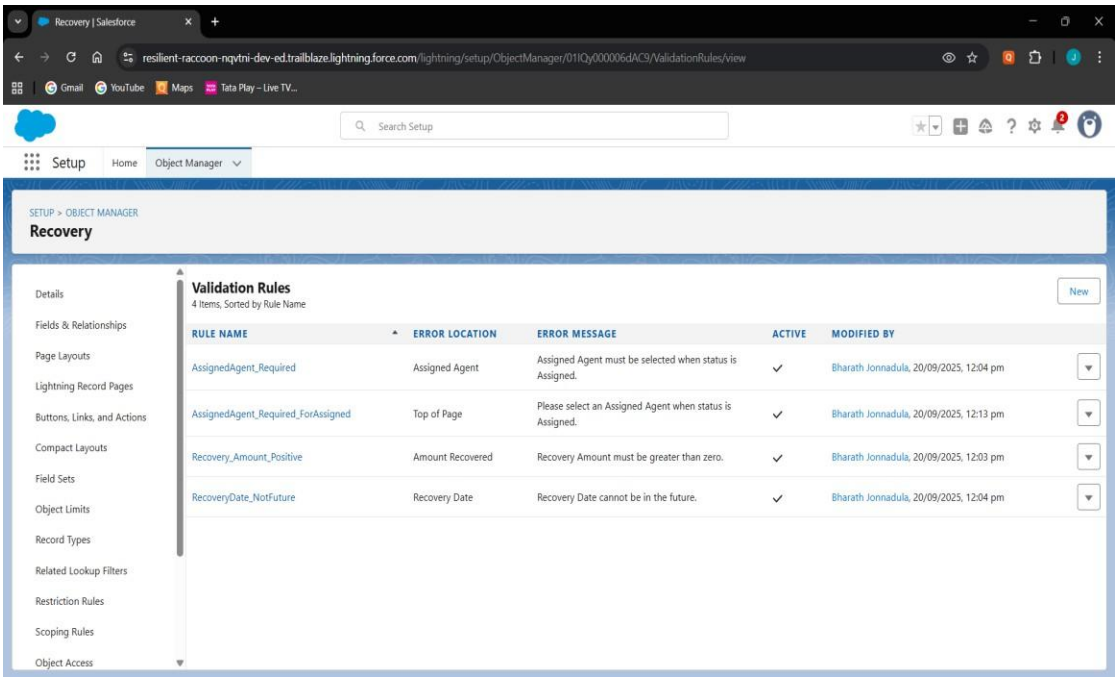
Phase 4: Process Automation (Admin)

Validation Rules

- Ensure values are valid (Loan/Payment/Recovery > 0).
- Prevent negative amounts and invalid dates.

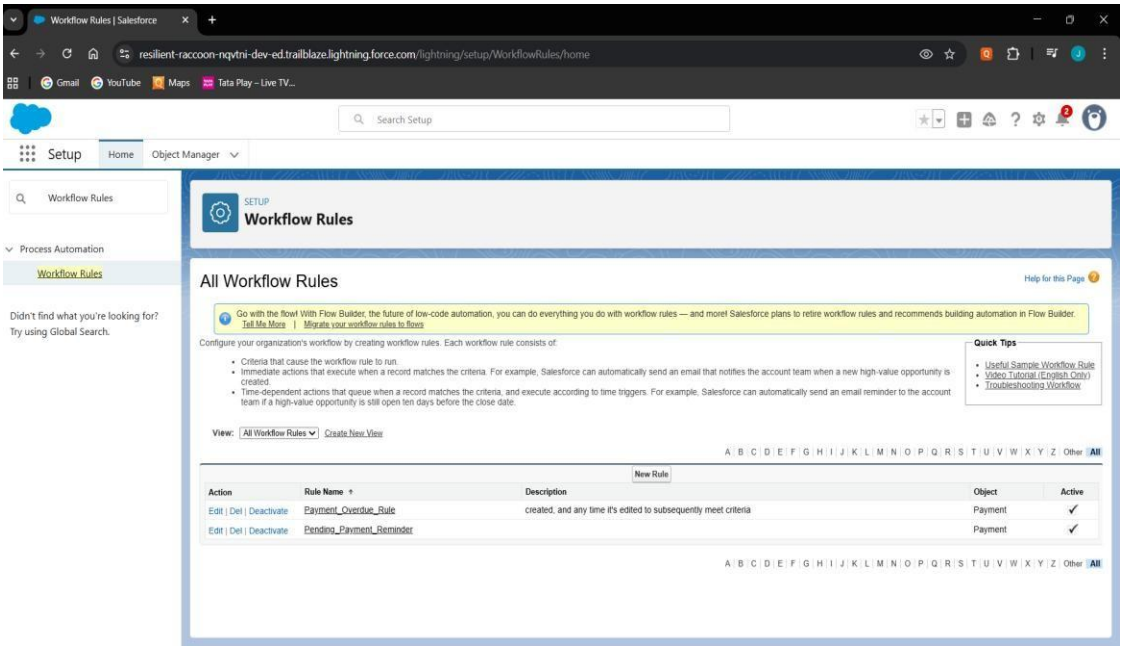






Workflow Rules

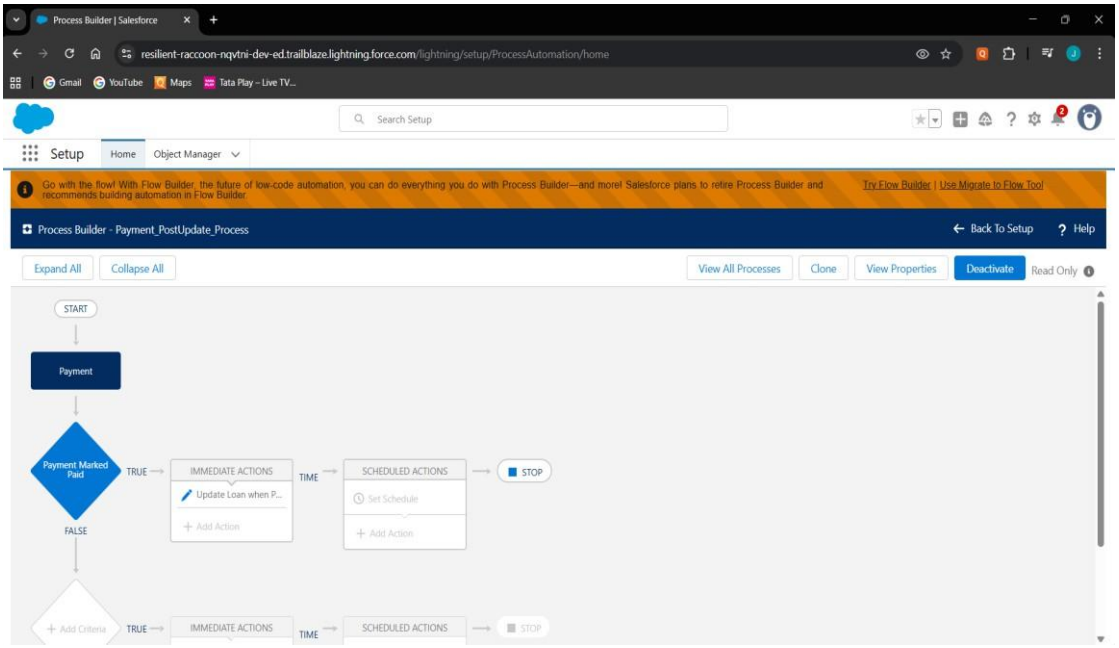
- Rule: Pending_Payment_Reminder.
- Sends reminder email for pending payments.

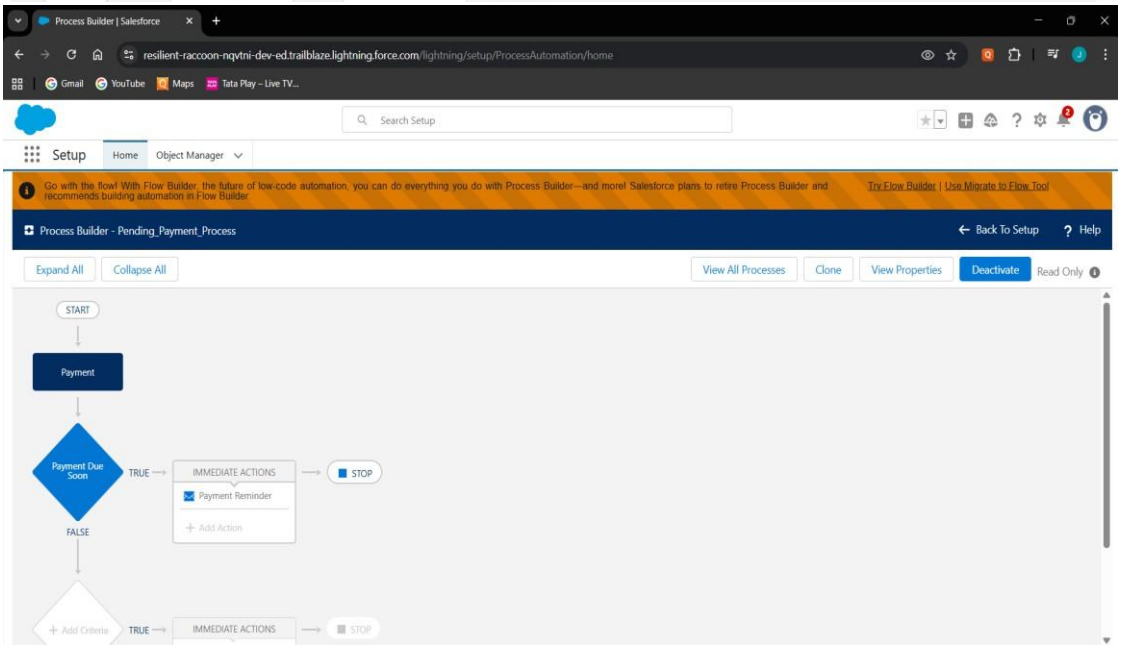


Process Builder

- Sends email reminders for due payments.
- Updates Loan status when all Payments are paid.

- Creates follow-up task for overdue payments.





Approval Process

- Loans > 5 Lakhs require Manager approval.
- Auto email + status update on approval/rejection.

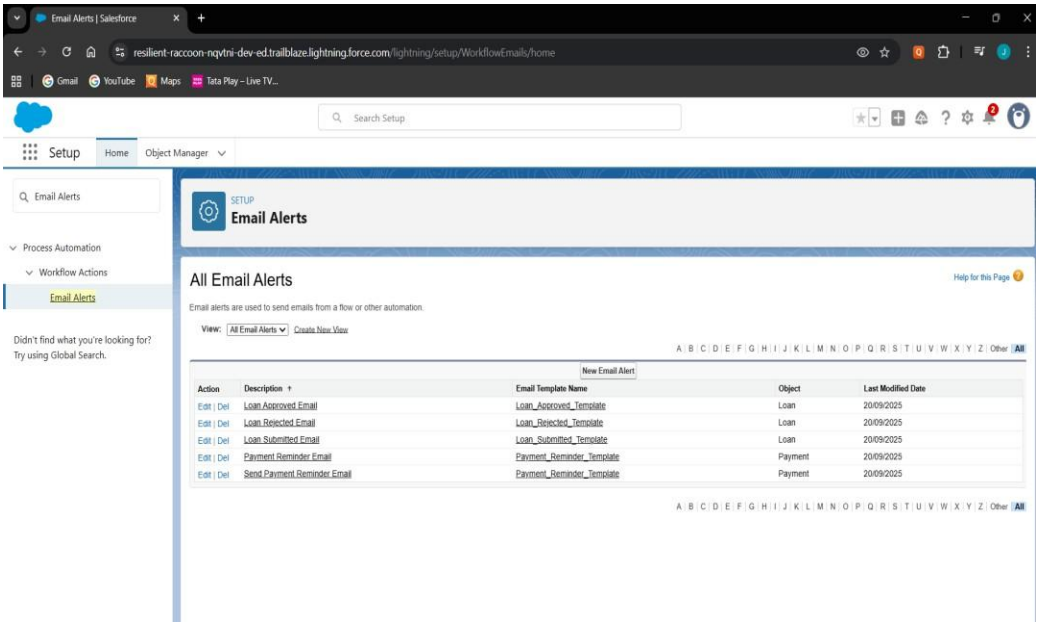
Flow Builder

- Scheduled Flow: Sends payment reminders 2 days before due.
- Record-Triggered: Marks payments overdue automatically.

- Record-Triggered: Creates Recovery record when overdue.

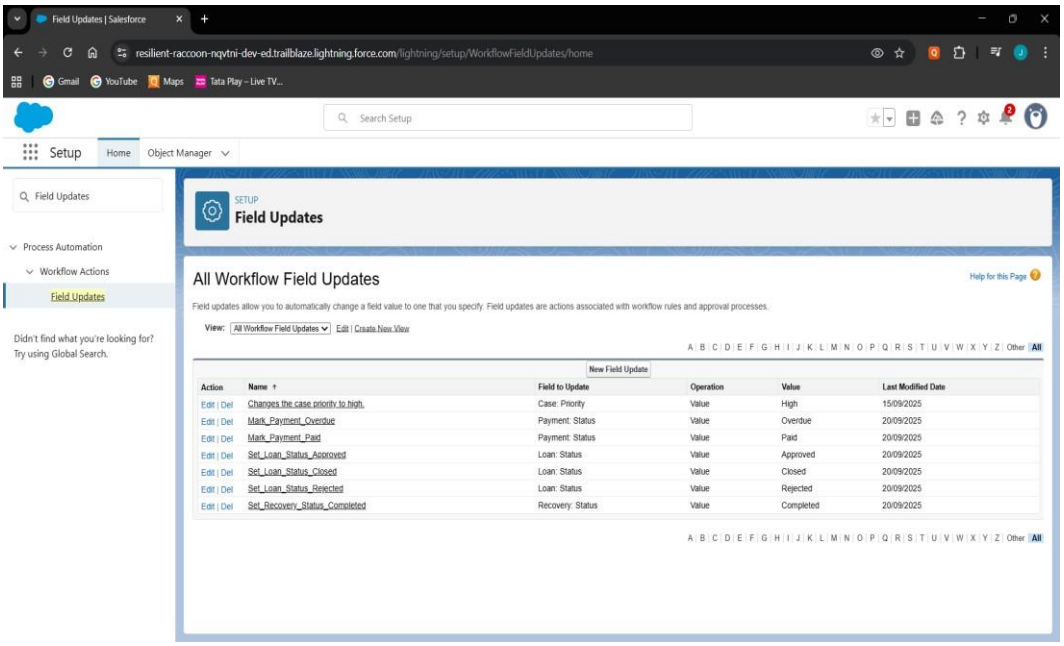
Email Alerts

- Payment_Reminder_Email → Customer.
- Loan_Submitted, Approved, Rejected → Manager/Officer.



Field Updates

- Auto-change Payment.Status to Overdue.
- Auto-close Loan when all Payments are Paid.



Tasks

- Task created when Payment is Overdue.
- Assigned to Recovery Agent with high priority.

Custom Notifications

- Sends in-app notification for Overdue Payments.

Phase 5: Apex Programming (Developer)

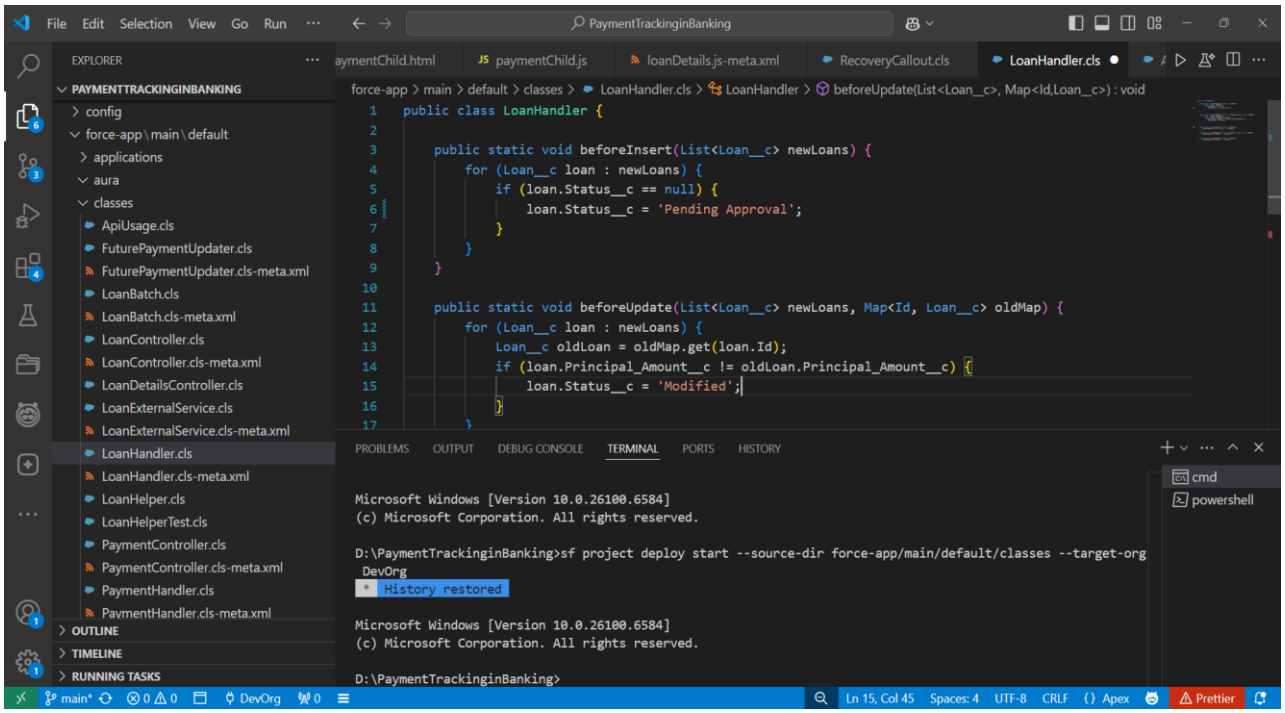
Classes & Objects

- Created Apex helper classes (LoanHandler, PaymentHandler, RecoveryHandler) to separate business logic from triggers.
- Used object-oriented approach for cleaner, reusable, and modular code.

LoanHandler.cls

```
public class LoanHandler {

    public static void beforeInsert(List<Loan__c>
newLoans) {
        for (Loan__c loan : newLoans) {
            if (loan.Status__c == null) {
                loan.Status__c = 'Pending Approval';
            }
        }
    }

    public static void beforeUpdate(List<Loan__c>
newLoans, Map<Id, Loan__c> oldMap) {
        for (Loan__c loan : newLoans) {
            Loan__c oldLoan = oldMap.get(loan.Id);
            if (loan.Principal_Amount__c !=
oldLoan.Principal_Amount__c) {
                loan.Status__c = 'Modified';
            }
        }
    }

    public static void afterInsert(List<Loan__c> newLoans)
{
        System.debug('New Loans inserted: ' + newLoans);
    }
```

```
    public static void afterUpdate(List<Loan__c>
newLoans, Map<Id, Loan__c> oldMap) {
        System.debug('Loans updated: ' + newLoans);
    }

    public static void afterDelete(List<Loan__c> oldLoans)
{
        System.debug('Loans deleted: ' + oldLoans);
    }
}
```

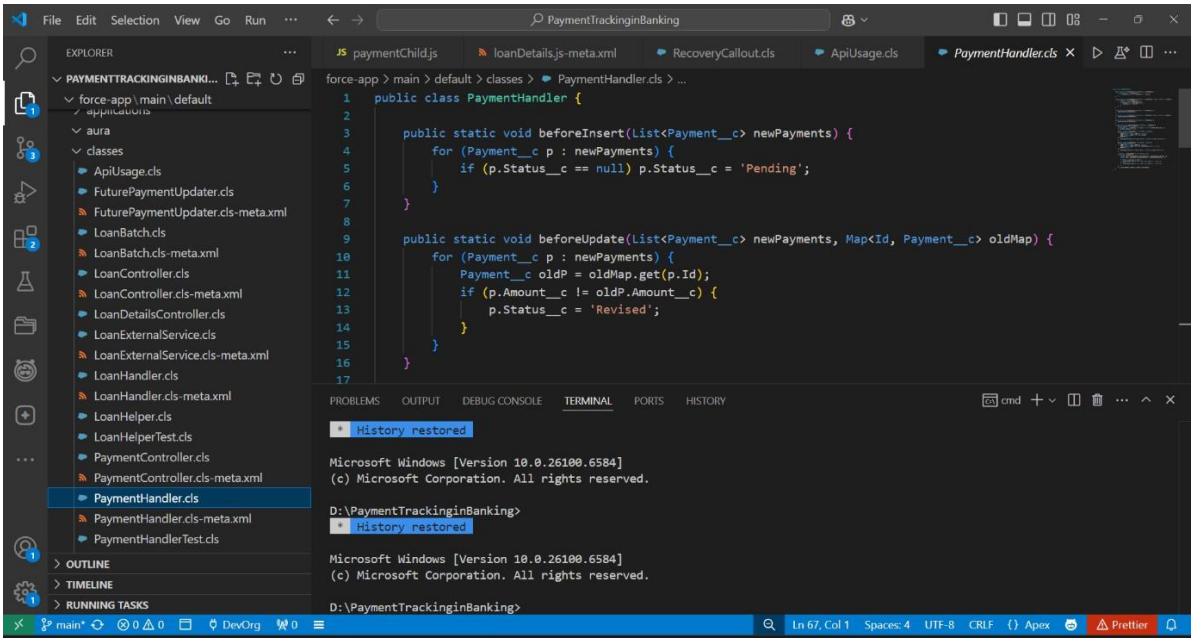


PaymentHandler.cls

```
public class PaymentHandler {

    public static void beforeInsert(List<Payment__c> newPayments) {
        for (Payment__c p : newPayments) {
            if (p.Status__c == null) p.Status__c = 'Pending';
        }
    }

    public static void beforeUpdate(List<Payment__c> newPayments,
     Map<Id, Payment__c> oldMap) {
        for (Payment__c p : newPayments) {
            Payment__c oldP = oldMap.get(p.Id);
            if (p.Amount__c != oldP.Amount__c) {
                p.Status__c = 'Revised';
            }
        }
    }

    public static void afterInsert(List<Payment__c> newPayments) {
        updateLoanStatus(newPayments);
    }

    public static void afterUpdate(List<Payment__c> newPayments,
     Map<Id, Payment__c> oldMap) {
        updateLoanStatus(newPayments);
    }

    public static void afterDelete(List<Payment__c> oldPayments) {
        updateLoanStatus(oldPayments);
    }

    private static void updateLoanStatus(List<Payment__c> payments) {
        Set<Id> loanIds = new Set<Id>();
        for (Payment__c p : payments) if (p.Loan__c != null)
     loanIds.add(p.Loan__c);
        if (loanIds.isEmpty()) return;
```

```
  Map<Id, Integer> totalPayments = new Map<Id, Integer>();
  for (AggregateResult ar : [
     SELECT Loan__c loanId, COUNT(Id) totalCount
     FROM Payment__c WHERE Loan__c IN :loanIds
     GROUP BY Loan__c
  ]) {
     totalPayments.put((Id) ar.get('loanId'), (Integer)
ar.get('totalCount'));
  }

  Map<Id, Integer> paidPayments = new Map<Id, Integer>();
  for (AggregateResult ar : [
     SELECT Loan__c loanId, COUNT(Id) paidCount
     FROM Payment__c WHERE Loan__c IN :loanIds AND Status__c
= 'Paid'
     GROUP BY Loan__c
  ]) {
     paidPayments.put((Id) ar.get('loanId'), (Integer)
ar.get('paidCount'));
  }

  List<Loan__c> loansToUpdate = new List<Loan__c>();
  for (Id lid : loanIds) {
     Integer total = totalPayments.containsKey(lid) ?
totalPayments.get(lid) : 0;
     Integer paid = paidPayments.containsKey(lid) ?
paidPayments.get(lid) : 0;

     if (total > 0 && total == paid) {
        loansToUpdate.add(new Loan__c(Id = lid, Status__c =
'Closed'));
     } else {
        loansToUpdate.add(new Loan__c(Id = lid, Status__c =
'Active'));
     }
  }
  if (!loansToUpdate.isEmpty()) update loansToUpdate;
```

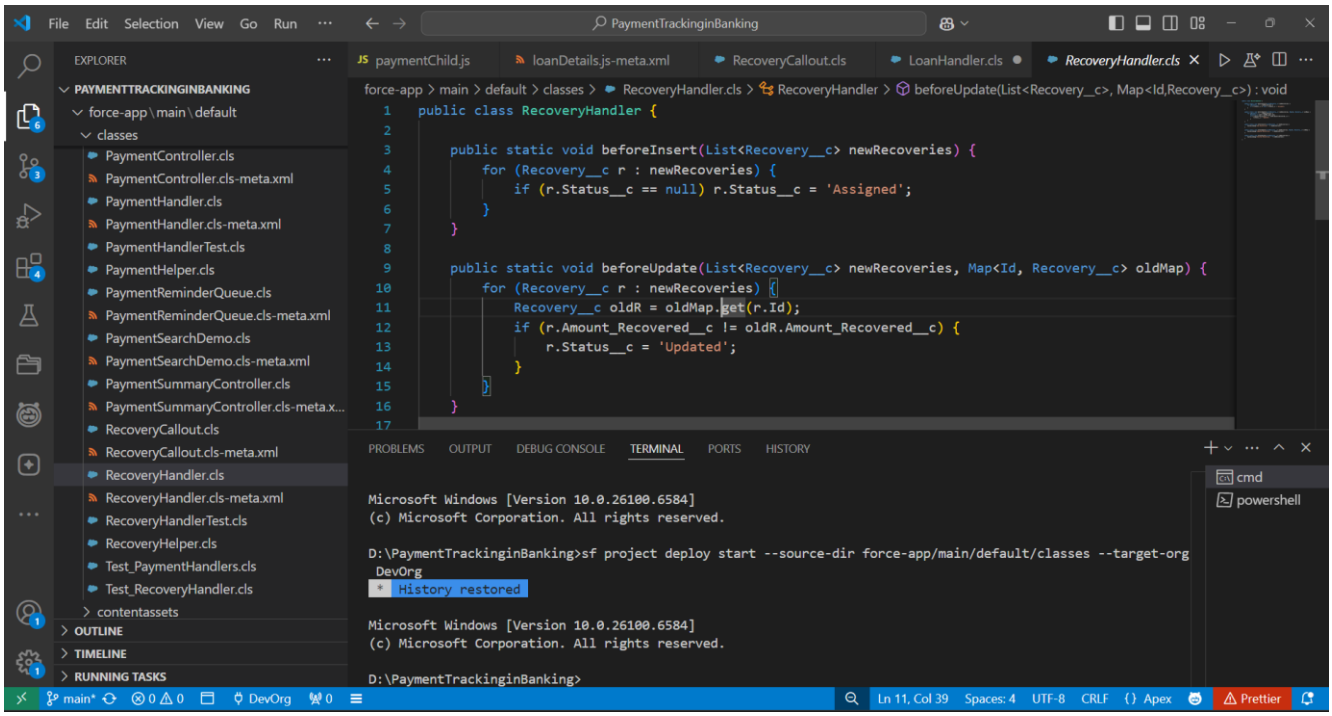
```
    }
}
```



RecoveryHandler.cls

```
public class RecoveryHandler {

    public static void beforeInsert(List<Recovery__c> newRecoveries) {
        for (Recovery__c r : newRecoveries) {
            if (r.Status__c == null) r.Status__c = 'Assigned';
        }
    }

    public static void beforeUpdate(List<Recovery__c> newRecoveries,
     Map<Id, Recovery__c> oldMap) {
        for (Recovery__c r : newRecoveries) {
            Recovery__c oldR = oldMap.get(r.Id);
            if (r.Amount_Recovered__c != oldR.Amount_Recovered__c) {
                r.Status__c = 'Updated';
            }
        }
    }
```

```
    public static void afterInsert(List<Recovery__c> newRecoveries) {
        System.debug('New Recoveries: ' + newRecoveries);
    }

    public static void afterUpdate(List<Recovery__c> newRecoveries,
     Map<Id, Recovery__c> oldMap) {
        System.debug('Updated Recoveries: ' + newRecoveries);
    }

    public static void afterDelete(List<Recovery__c> oldRecoveries) {
        System.debug('Deleted Recoveries: ' + oldRecoveries);
    }
}
```

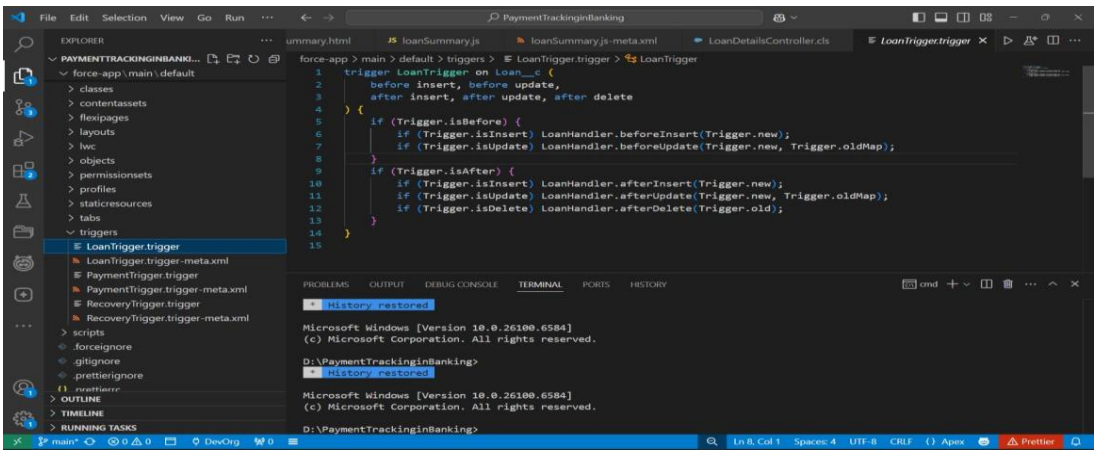


Apex Triggers

- Implemented triggers for Loan, Payment, and Recovery to handle automation like updating statuses and roll-ups.

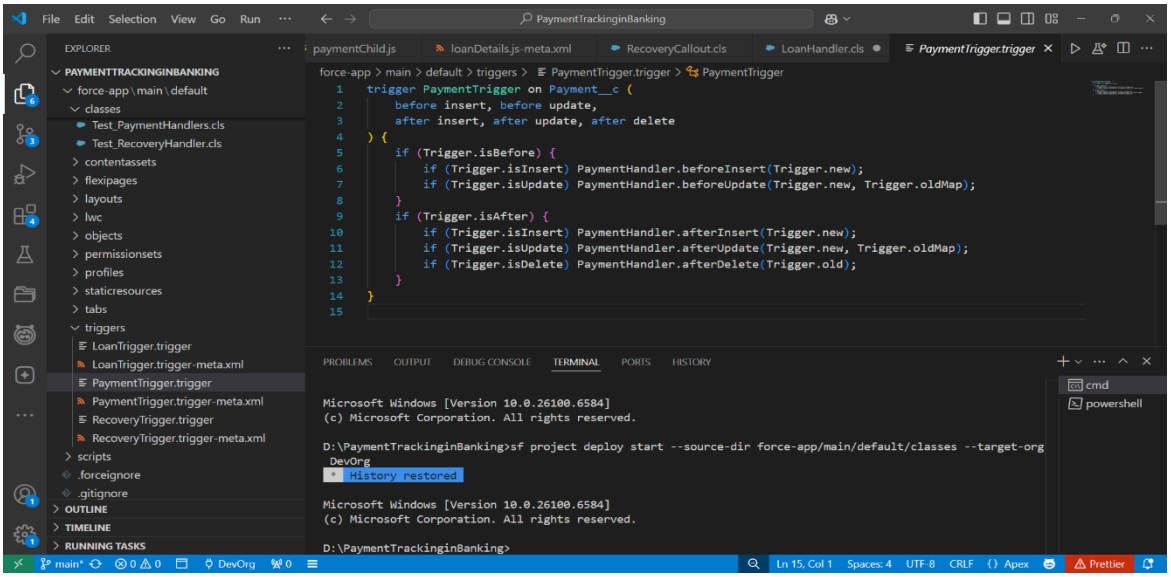
LoanTrigger.trigger

```
trigger LoanTrigger on Loan__c (
    before insert, before update,
    after insert, after update, after delete
) {
    if (Trigger.isBefore) {
        if (Trigger.isInsert) LoanHandler.beforeInsert(Trigger.new);
        if (Trigger.isUpdate) LoanHandler.beforeUpdate(Trigger.new,
    Trigger.oldMap);
    }
    if (Trigger.isAfter) {
        if (Trigger.isInsert) LoanHandler.afterInsert(Trigger.new);
        if (Trigger.isUpdate) LoanHandler.afterUpdate(Trigger.new,
    Trigger.oldMap);
        if (Trigger.isDelete) LoanHandler.afterDelete(Trigger.old);}}}
```



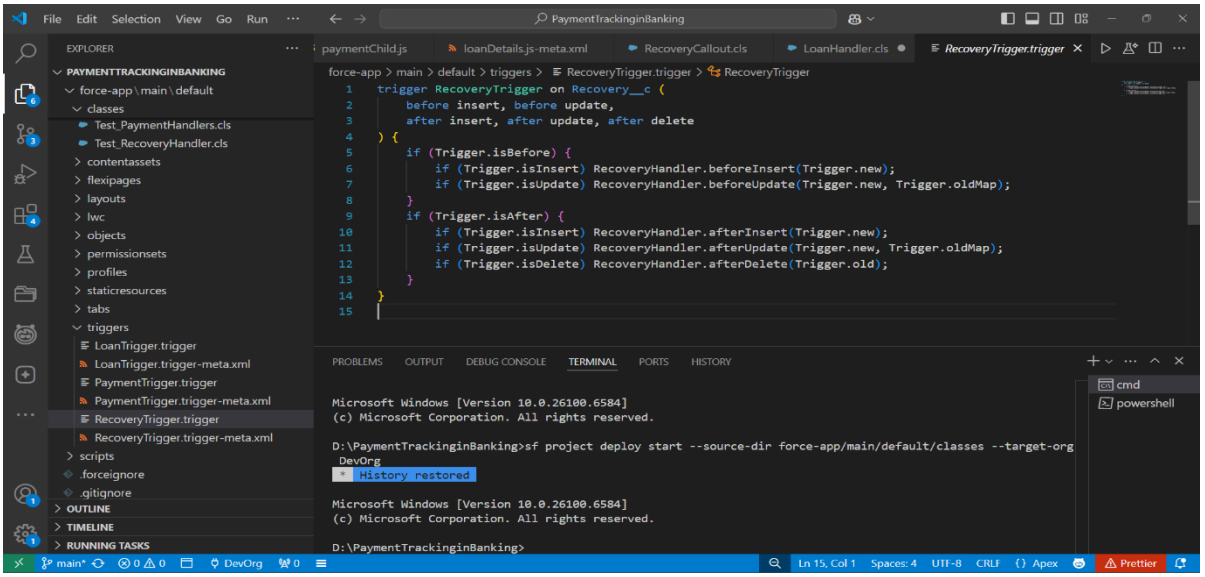
PaymentTrigger.trigger

```
trigger PaymentTrigger on Payment__c (
    before insert, before update,
    after insert, after update, after delete
) {
    if (Trigger.isBefore) {
        if (Trigger.isInsert) PaymentHandler.beforeInsert(Trigger.new);
        if (Trigger.isUpdate)
     PaymentHandler.beforeUpdate(Trigger.new, Trigger.oldMap);
    }
    if (Trigger.isAfter) {
        if (Trigger.isInsert) PaymentHandler.afterInsert(Trigger.new);
        if (Trigger.isUpdate)
     PaymentHandler.afterUpdate(Trigger.new, Trigger.oldMap);
        if (Trigger.isDelete) PaymentHandler.afterDelete(Trigger.old);
    }
}
```



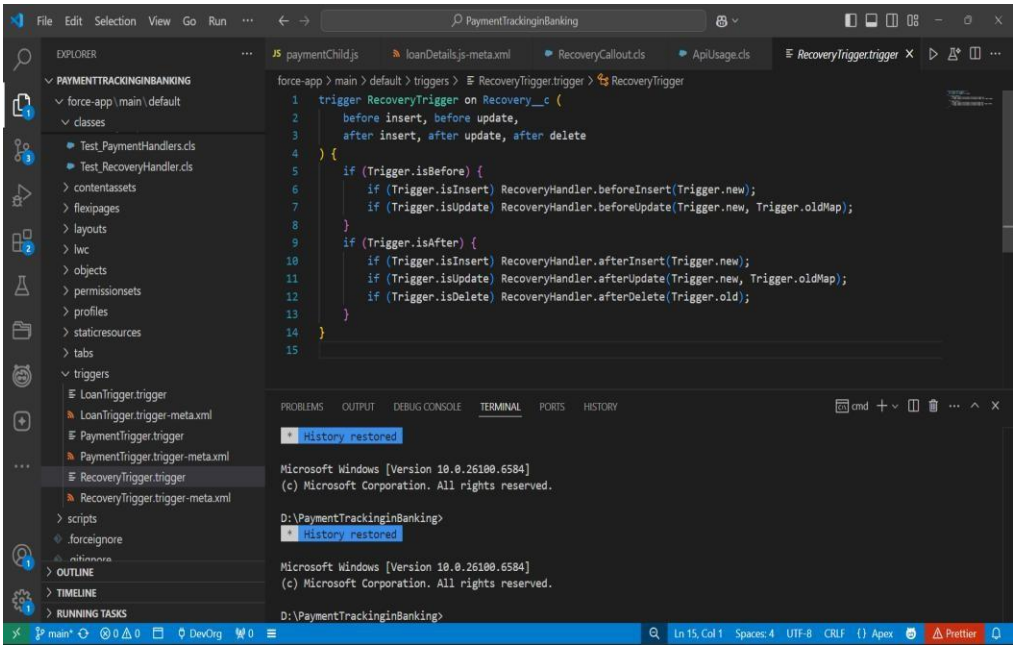
RecoveryTrigger.trigger

```
trigger RecoveryTrigger on Recovery__c (
    before insert, before update,
    after insert, after update, after delete
) {
    if (Trigger.isBefore) {
        if (Trigger.isInsert) RecoveryHandler.beforeInsert(Trigger.new);
        if (Trigger.isUpdate)
     RecoveryHandler.beforeUpdate(Trigger.new, Trigger.oldMap);
    }
    if (Trigger.isAfter) {
        if (Trigger.isInsert) RecoveryHandler.afterInsert(Trigger.new);
        if (Trigger.isUpdate)
     RecoveryHandler.afterUpdate(Trigger.new, Trigger.oldMap);
        if (Trigger.isDelete) RecoveryHandler.afterDelete(Trigger.old);
    }}
```



Trigger Design Pattern

- Applied Trigger Handler Pattern to keep trigger code clean and delegate logic to handler classes.
- Ensured only one trigger per object with proper event handling.



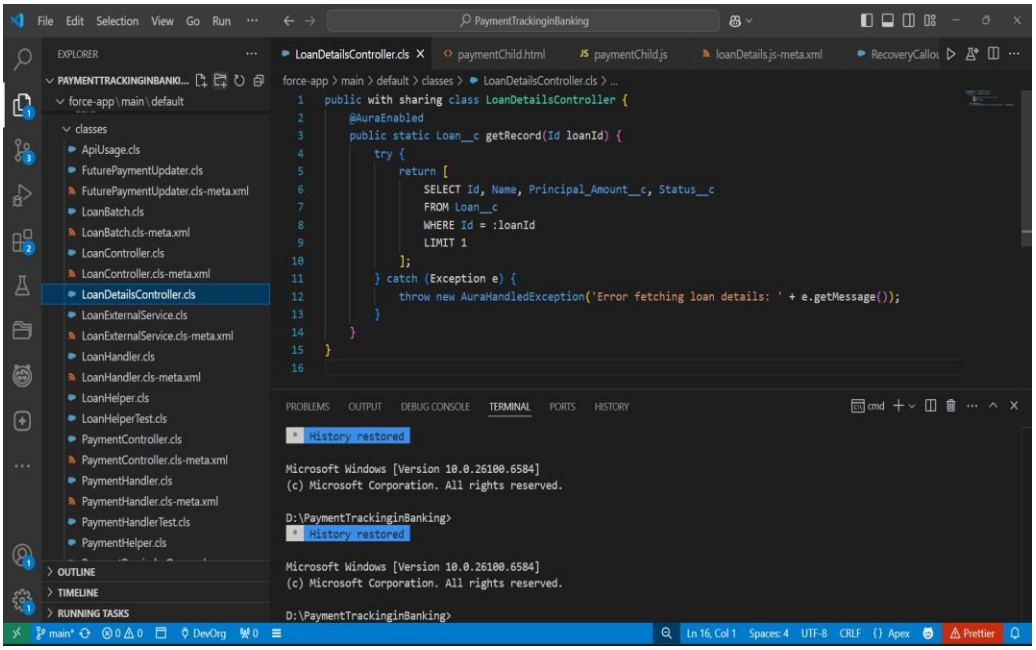
SOQL & SOSL

- Used SOQL in classes to query Loans, Payments, and Recoveries.
- Implemented simple search functionality to fetch customer records.

LoanDetailsController.cls

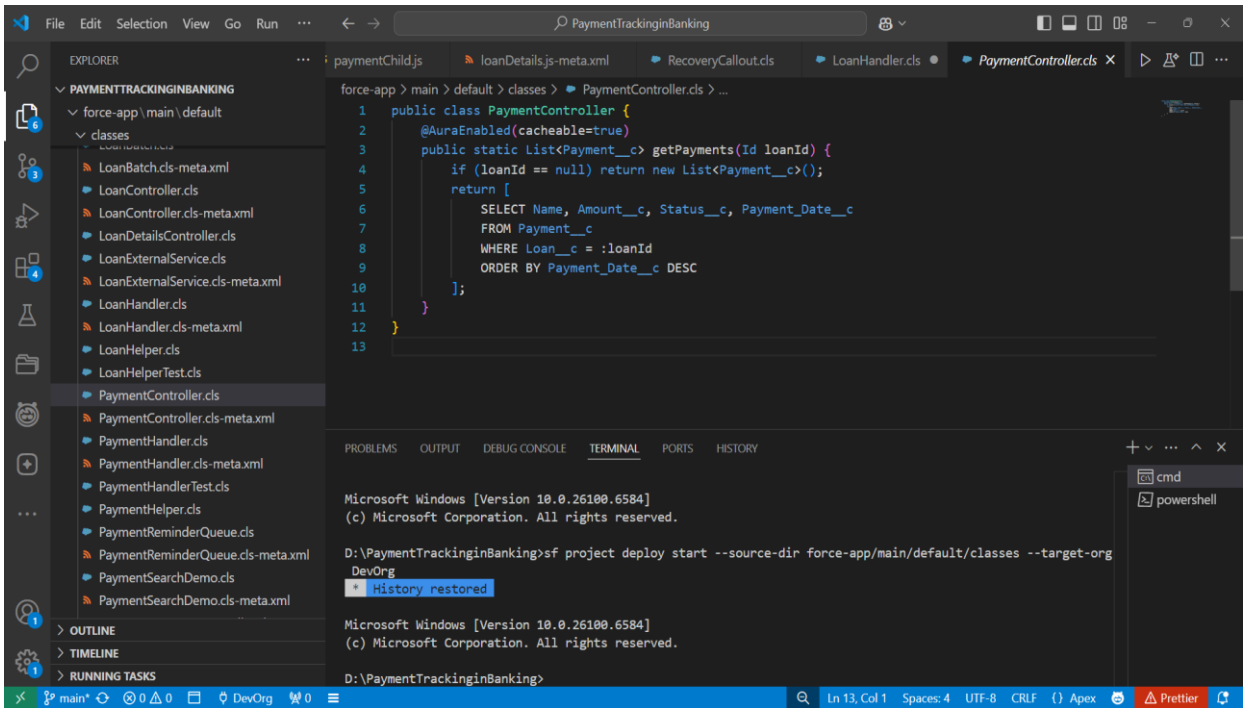
```
public with sharing class LoanController {
@AuraEnabled(cacheable=true)
public static Loan__c getLoan(String loanId) {
 return [SELECT  Name, Principal_Amount__c FROM
Loan__c WHERE Id = :loanId LIMIT 1];
}
```

```
@AuraEnabled
   public static Boolean markLoanClosed(String loanId) {
    try {
      Loan__c L = [SELECT Id, Status__c FROM Loan__c WHERE
   Id = :loanId LIMIT 1];
      L.Status__c = 'Closed';
      update L;
      return true;
    } catch (Exception e) {
      throw new AuraHandledException('Cannot close loan: ' +
   e.getMessage());
    }
  }
}
```



PaymentsController.cls

```
public class PaymentController {
    @AuraEnabled(cacheable=true)
    public static List<Payment__c> getPayments(Id loanId) {
        if (loanId == null) return new List<Payment__c>();
        return [
            SELECT Name, Amount__c, Status__c,
     Payment_Date__c
            FROM Payment__c
            WHERE Loan__c = :loanId
            ORDER BY Payment_Date__c DESC
];
}
}
```

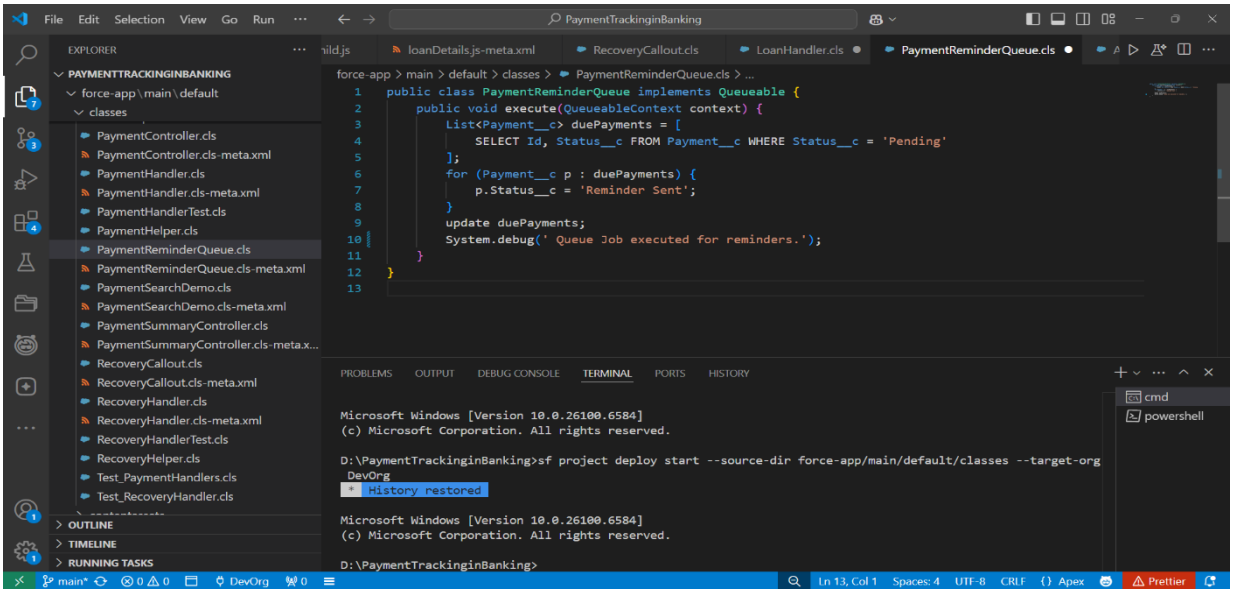


Collections: List, Set, Map

- Applied Lists to hold multiple Payments for bulk processing.
- Used Maps for Loan → Payments mapping to handle roll-ups efficiently.

PaymentRemainderQueue.cls

```
public class PaymentReminderQueue implements
Queueable {
public void execute(QueueableContext context) {
   List<Payment__c> duePayments = [
      SELECT Id, Status__c FROM Payment__c WHERE
Status__c = 'Pending'
   ];
   for (Payment__c p : duePayments) {
      p.Status__c = 'Reminder Sent';
   }
   update duePayments;
   System.debug(' Queue Job executed for reminders.');
  }
}
```



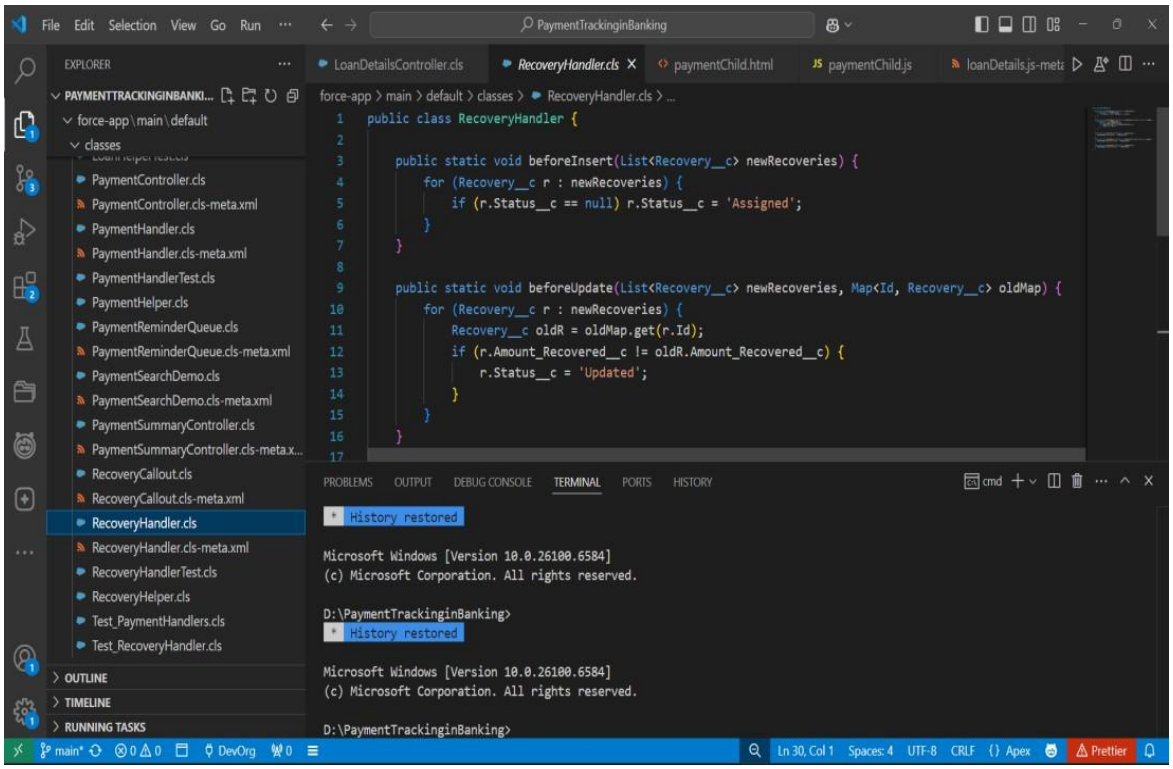
Control Statements

- Implemented IF conditions for validation.

RecoveryHandler.cls

```
public class RecoveryHandler {
    public static void beforeInsert(List<Recovery__c> newRecoveries) {
        for (Recovery__c r : newRecoveries) {
            if (r.Status__c == null) r.Status__c = 'Assigned';
        }
    }

    public static void beforeUpdate(List<Recovery__c> newRecoveries,
     Map<Id, Recovery__c> oldMap) {
        for (Recovery__c r : newRecoveries) {
            Recovery__c oldR = oldMap.get(r.Id);
            if (r.Amount_Recovered__c != oldR.Amount_Recovered__c) {
                r.Status__c = 'Updated';
            }
        }
    }

    public static void afterInsert(List<Recovery__c> newRecoveries) {
        System.debug('New Recoveries: ' + newRecoveries);
    }
```

```
public static void afterUpdate(List<Recovery__c> newRecoveries,
    Map<Id, Recovery__c> oldMap) {
        System.debug('Updated Recoveries: ' + newRecoveries);
    }

    public static void afterDelete(List<Recovery__c> oldRecoveries) {
        System.debug('Deleted Recoveries: ' + oldRecoveries);
    }
}
```

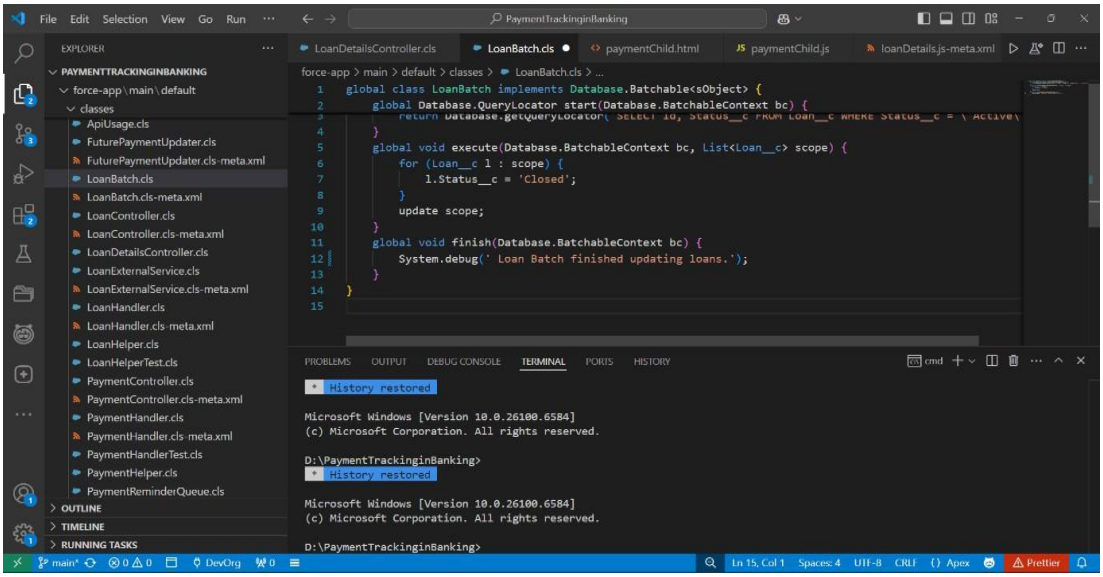


Batch Apex

- Created simple batch class for recalculating loan balances.
- Enabled processing of large datasets asynchronously.

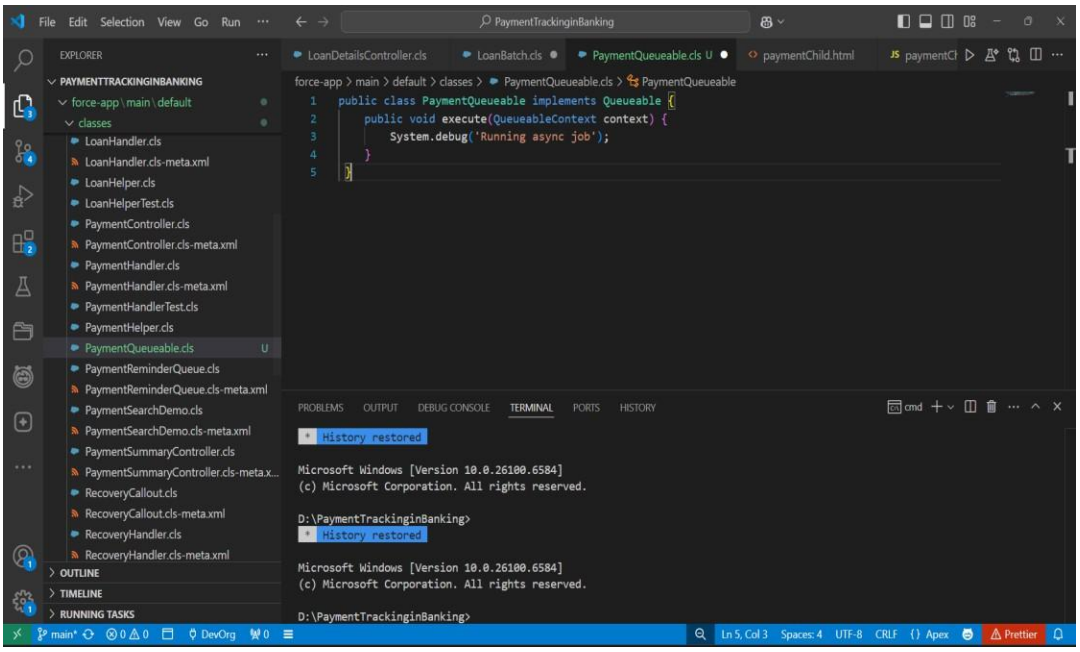
LoanBatch.cls

```
global class LoanBatch implements Database.Batchable<sObject> {
global Database.QueryLocator start(Database.BatchableContext bc)
 {
   return Database.getQueryLocator('SELECT Id, Status__c FROM
 Loan__c WHERE Status__c = \'Active\'');
}
global void execute(Database.BatchableContext bc, List<Loan__c>
 scope) {
   for (Loan__c l : scope) {
      l.Status__c = 'Closed';
   }
   update scope;
}
global void finish(Database.BatchableContext bc) {
   System.debug(' Loan Batch finished updating loans.');
}
}
```



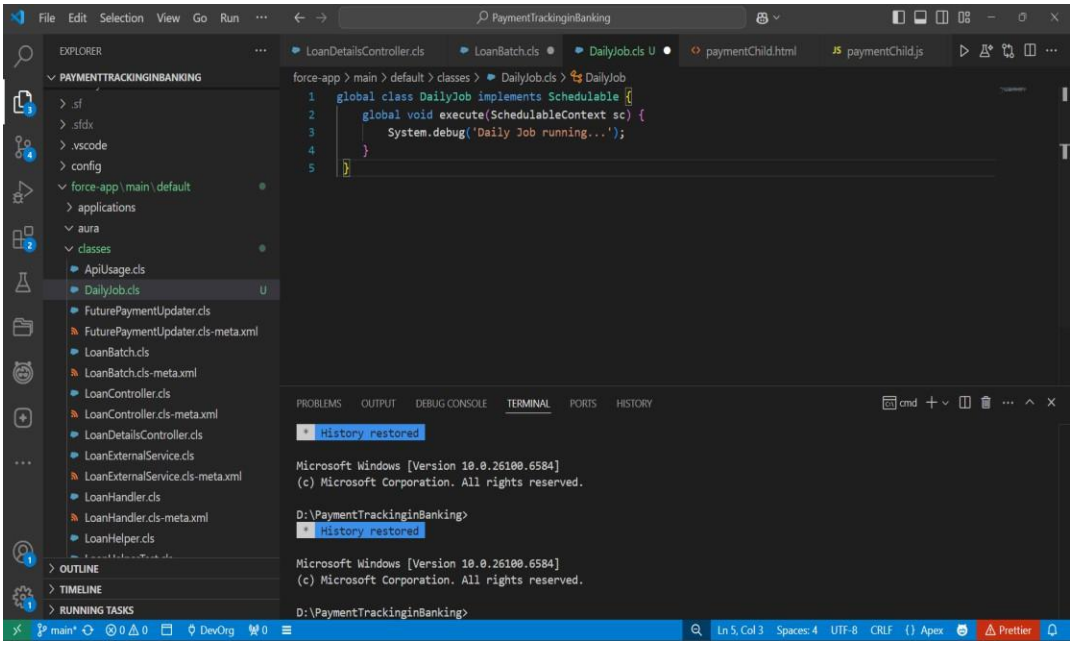
Queueable Apex

- Used Queueable Apex for lightweight async operations like notifications.
- Provides better chaining support compared to future methods.



Scheduled Apex

- Built a scheduler to run daily overdue payment checks.
- Automated reminders and recovery creation without manual intervention.



Future Methods

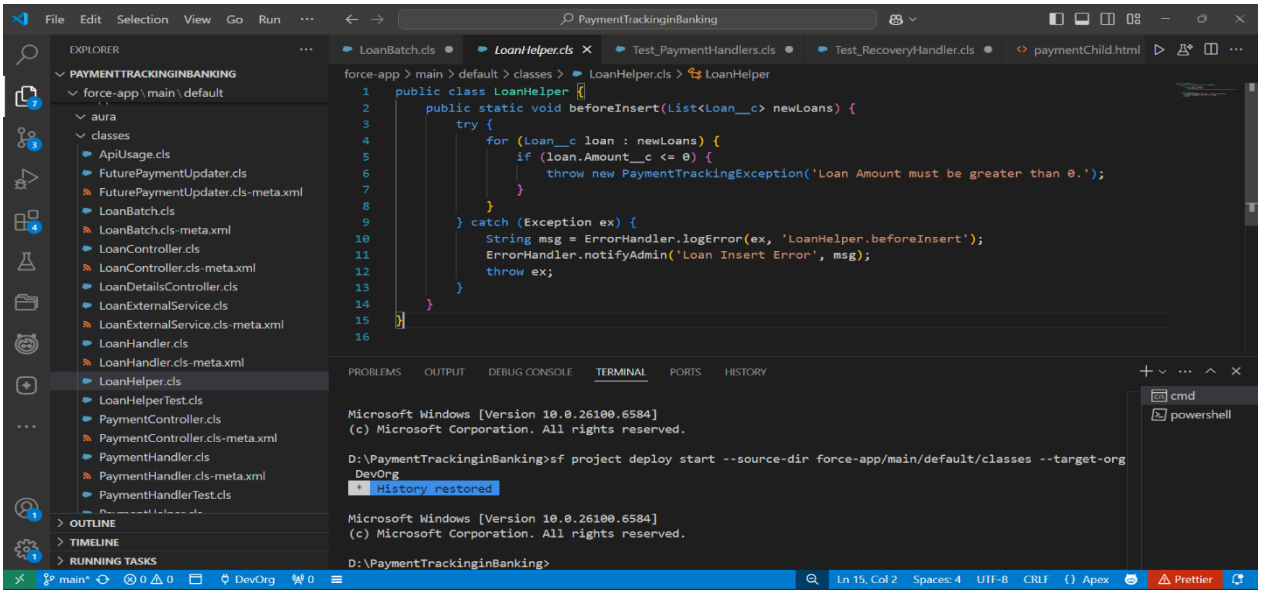
- Implemented future call for sending async email notifications.

Exception Handling

- Created a centralized ErrorHandler class to log and handle exceptions.
- Ensured smooth error reporting without stopping automation.

LoanHelper.cls

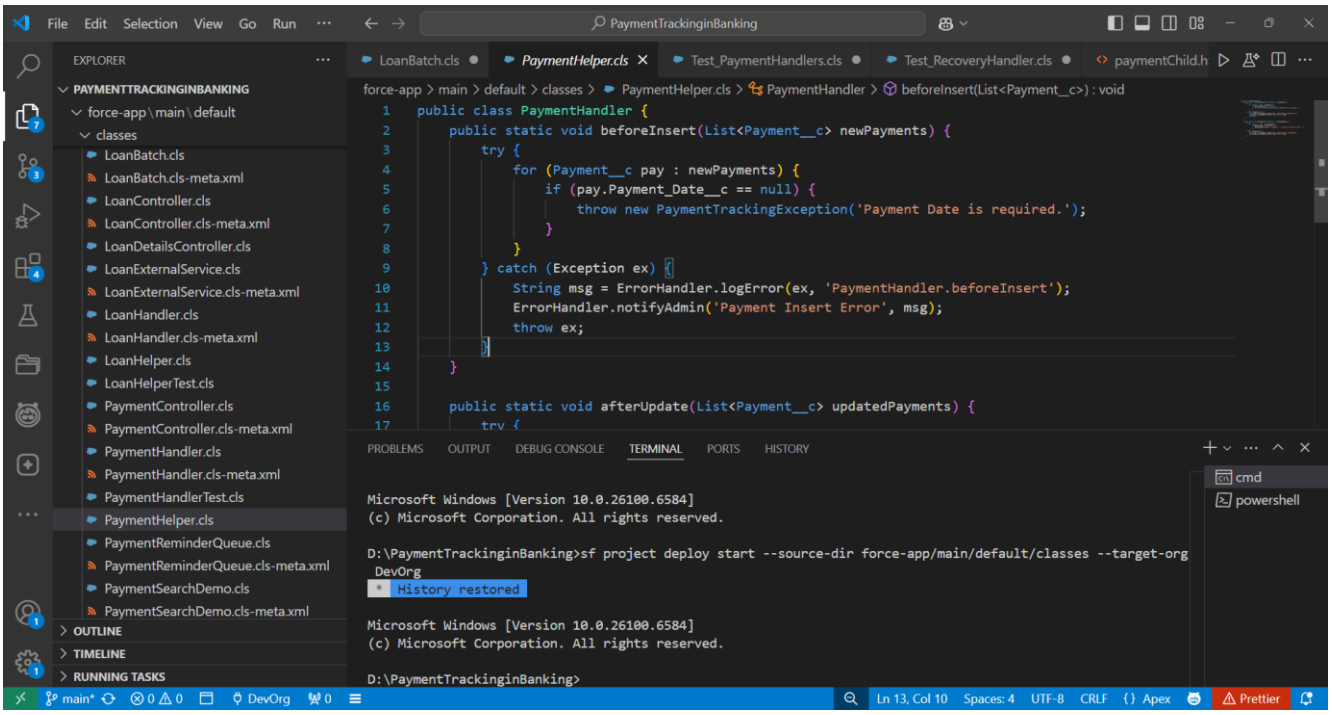
```
public class LoanHelper {
public static void beforeInsert(List<Loan__c> newLoans) {
  try {
    for (Loan__c loan : newLoans) {
      if (loan.Amount__c <= 0) {
        throw new PaymentTrackingException('Loan Amount must be greater than 0.');
      }
    }
  } catch (Exception ex) {
    String msg = ErrorHandler.logError(ex, 'LoanHelper.beforeInsert');
    ErrorHandler.notifyAdmin('Loan Insert Error', msg);
    throw ex;
  }
 }
}
```



PaymentHelper.cls

```
public class PaymentHandler {
   public static void beforeInsert(List<Payment__c> newPayments) {
      try {
         for (Payment__c pay : newPayments) {
            if (pay.Payment_Date__c == null) {
               throw new PaymentTrackingException('Payment Date is
    required.');
            }
         }
      } catch (Exception ex) {
         String msg = ErrorHandler.logError(ex,
    'PaymentHandler.beforeInsert');
         ErrorHandler.notifyAdmin('Payment Insert Error', msg);
         throw ex;
      }
   }

   public static void afterUpdate(List<Payment__c> updatedPayments)
    {
      try {
```

```
        for (Payment__c pay : updatedPayments) {
            if (pay.Status__c == 'Overdue') {
                System.debug('Payment ' + pay.Id + ' is overdue. Creating
    recovery...');
            }
        }
      } catch (Exception ex) {
        String msg = ErrorHandler.logError(ex,
    'PaymentHandler.afterUpdate');
        ErrorHandler.notifyAdmin('Payment Update Error', msg);
      }
    }
}
```

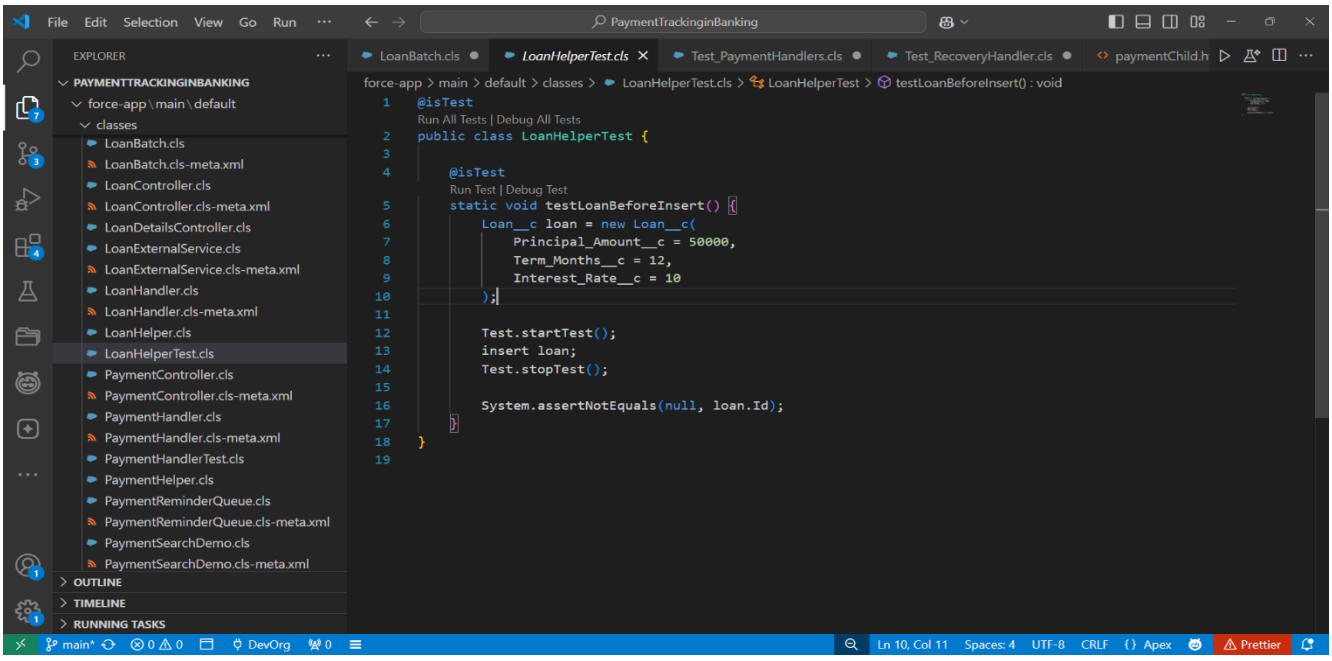


Test Classes

- Wrote test classes for each handler & trigger to validate functionality.

LoanHelperTest.cls

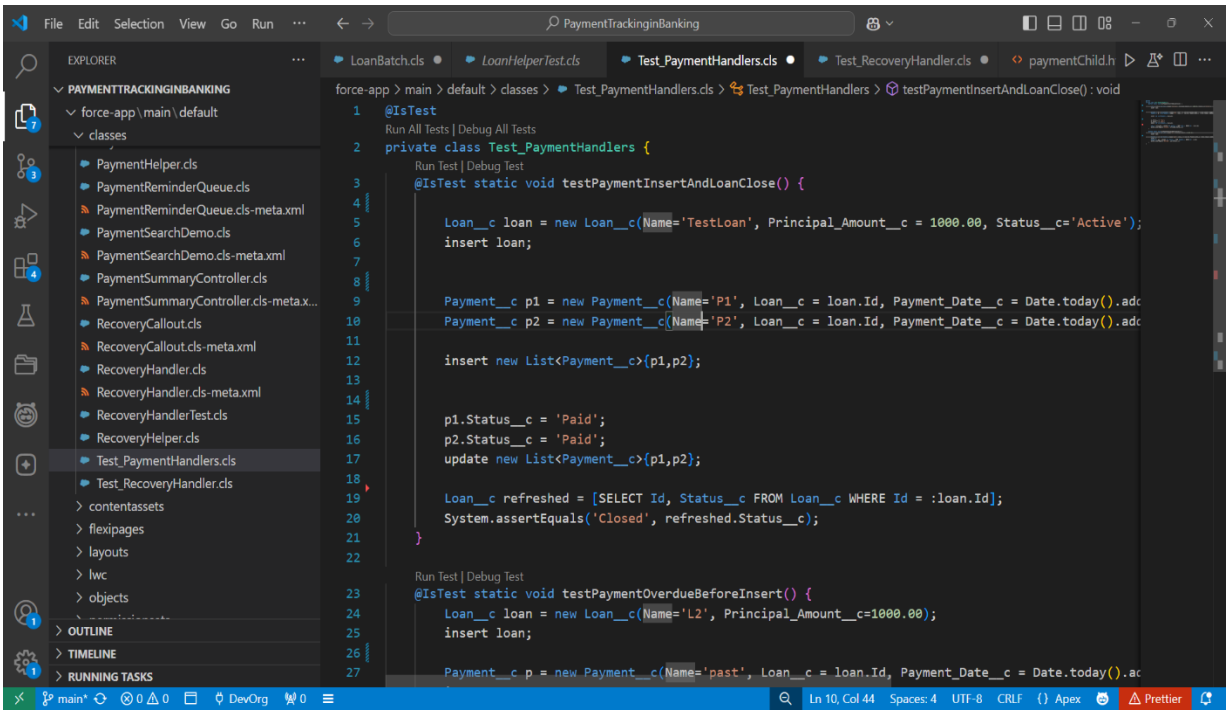
```
@isTest
public class LoanHelperTest {

    @isTest
    static void testLoanBeforeInsert() {
        Loan__c loan = new Loan__c(
            Principal_Amount__c = 50000,
            Term_Months__c = 12,
            Interest_Rate__c = 10
        );

        Test.startTest();
        insert loan;
        Test.stopTest();

        System.assertNotEquals(null, loan.Id);
    }
}
```



Test_PaymentHandlers.cls

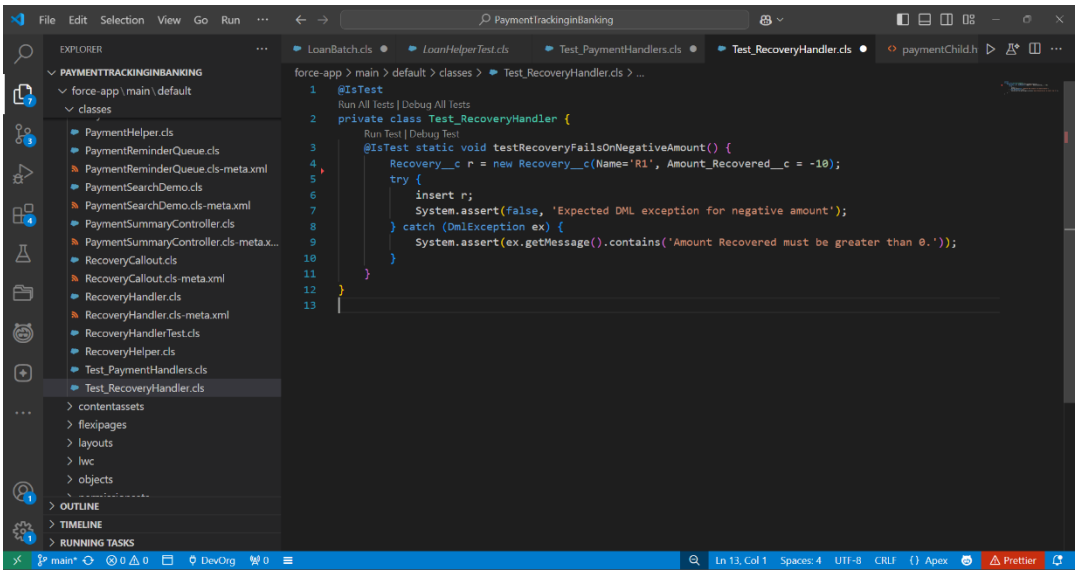
```
@IsTest
private class Test_PaymentHandlers {
   @IsTest static void testPaymentInsertAndLoanClose() {

      Loan__c loan = new Loan__c(Name='TestLoan',
    Principal_Amount__c = 1000.00, Status__c='Active');
      insert loan;


      Payment__c p1 = new Payment__c(Name='P1', Loan__c = loan.Id,
    Payment_Date__c = Date.today().addDays(1), Amount__c = 500,
    Status__c='Pending');
      Payment__c p2 = new Payment__c(Name='P2', Loan__c = loan.Id,
    Payment_Date__c = Date.today().addDays(2), Amount__c = 500,
    Status__c='Pending');

      insert new List<Payment__c>{p1,p2};


      p1.Status__c = 'Paid';
      p2.Status__c = 'Paid';
      update new List<Payment__c>{p1,p2};

      Loan__c refreshed = [SELECT Id, Status__c FROM Loan__c WHERE
    Id = :loan.Id];
      System.assertEquals('Closed', refreshed.Status__c);
   }

   @IsTest static void testPaymentOverdueBeforeInsert() {
      Loan__c loan = new Loan__c(Name='L2',
    Principal_Amount__c=1000.00);
      insert loan;
```

```
    Payment__c p = new Payment__c(Name='past', Loan__c =
  loan.Id, Payment_Date__c = Date.today().addDays(-2), Amount__c
  = 100, Status__c=null);
    insert p;
    Payment__c got = [SELECT Id, Status__c FROM Payment__c
  WHERE Id = :p.Id];
    System.assertEquals('Overdue', got.Status__c);
  }
}
```



Test_RecoveryHandlers.cls

```
@IsTest
private class Test_RecoveryHandler {
    @IsTest static void testRecoveryFailsOnNegativeAmount() {
        Recovery__c r = new Recovery__c(Name='R1',
    Amount_Recovered__c = -10);
        try {
            insert r;
            System.assert(false, 'Expected DML exception for negative
    amount');
        } catch (DmlException ex) {
            System.assert(ex.getMessage().contains('Amount Recovered
    must be greater than 0.'));
        }
    }
```




```
}
```

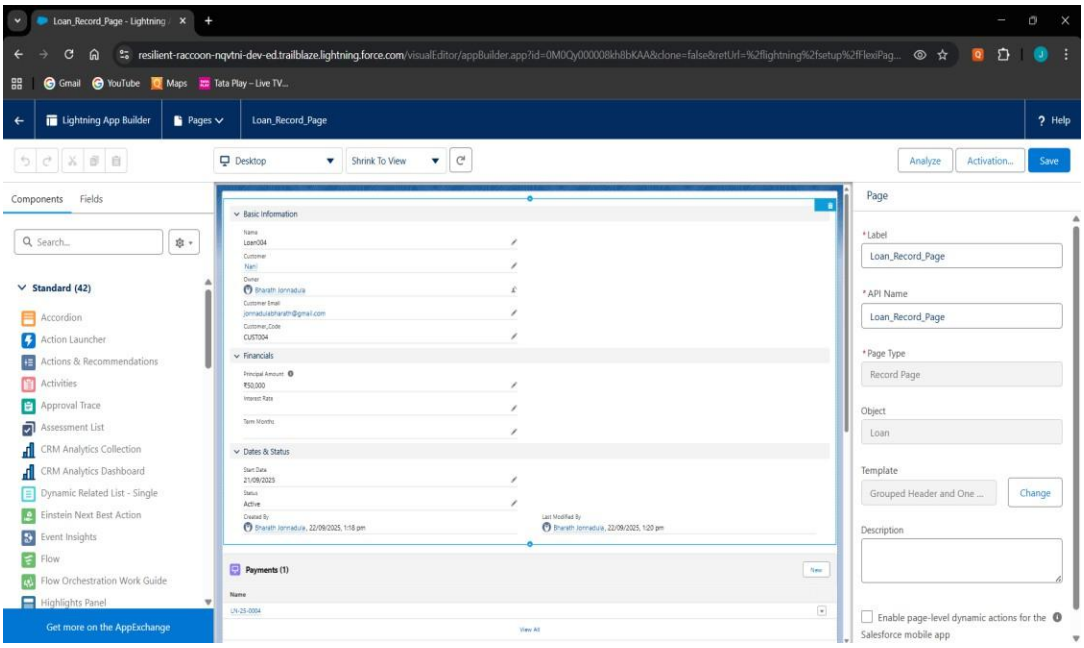
Asynchronous Processing

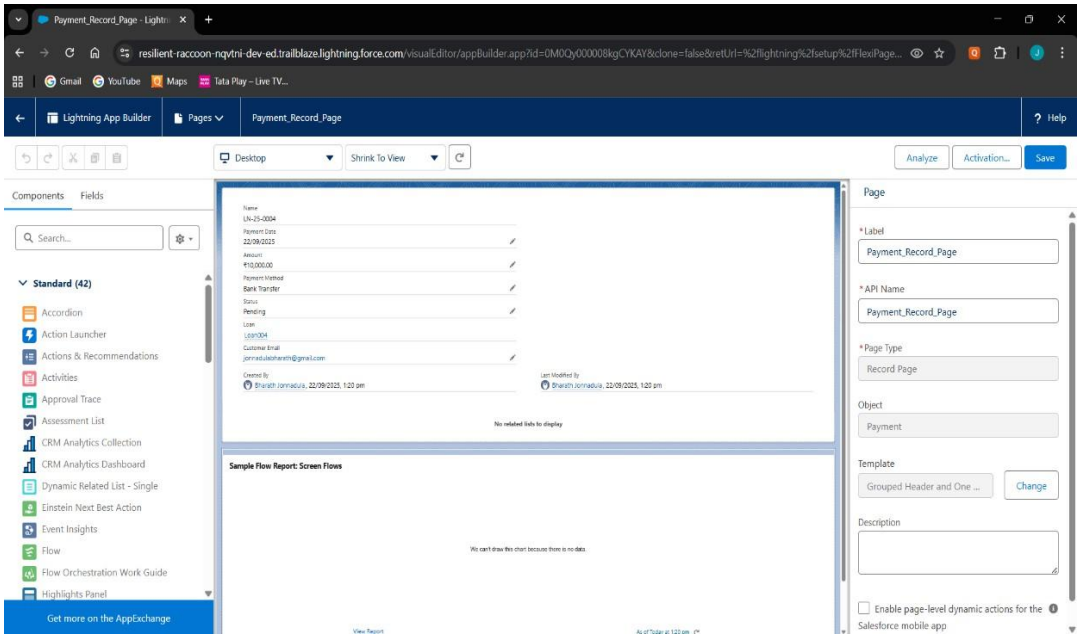
- Combined Batch, Queueable, Scheduled, and Future methods for async operations.

Phase 6: User Interface Development

Lightning App Builder

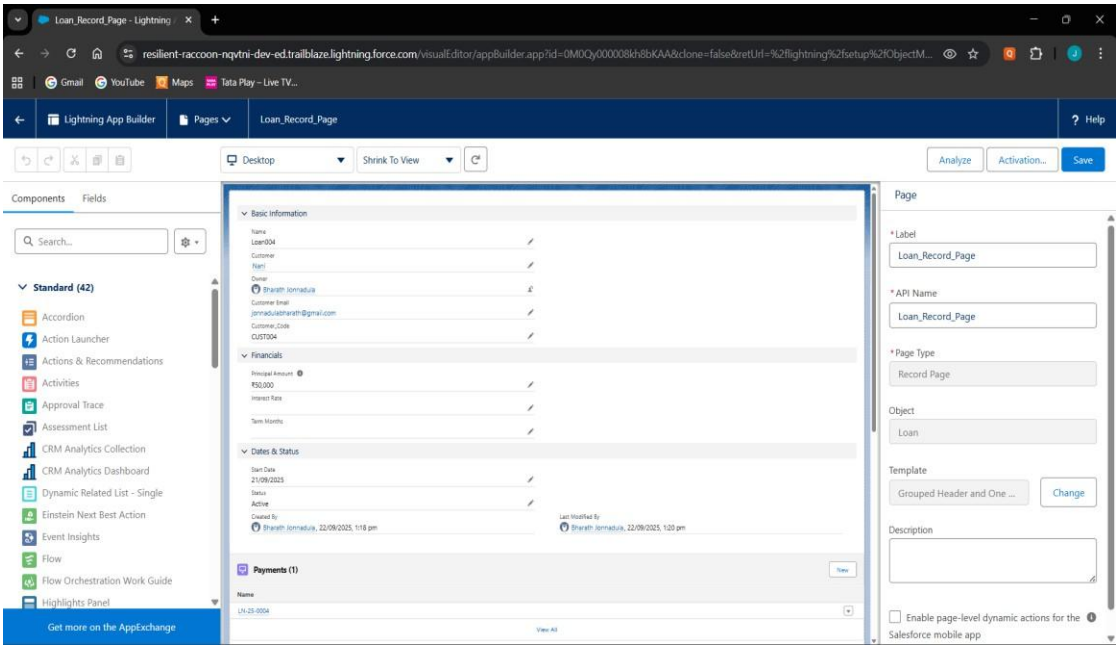
- Created a custom Loan_Record_Page, Payment_Home_Page, Payment_Record_Page, Recovery_record_Page

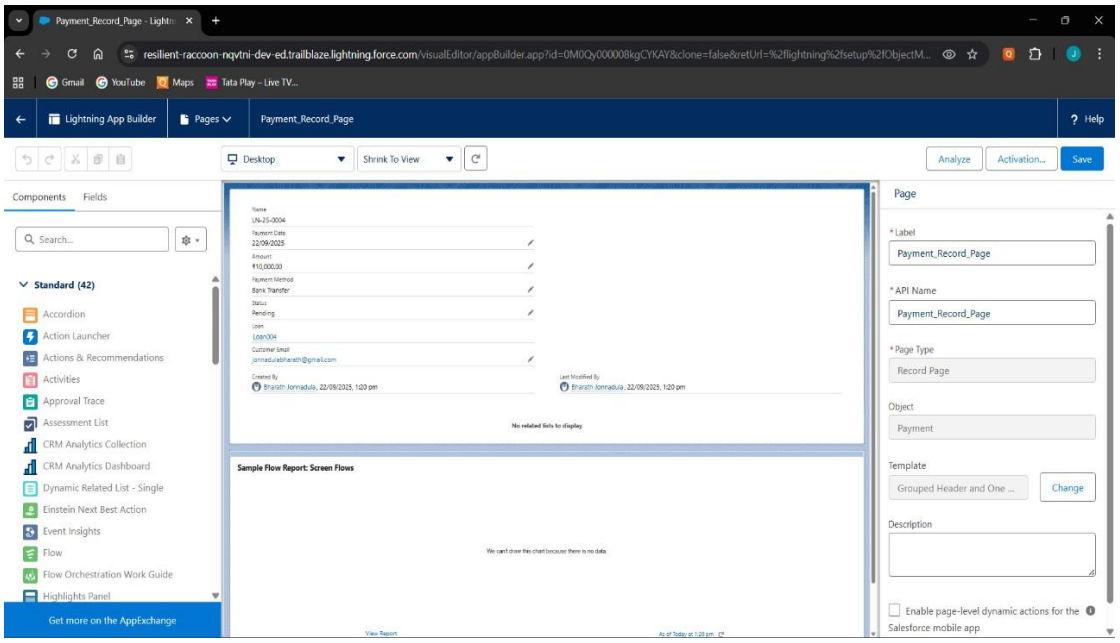


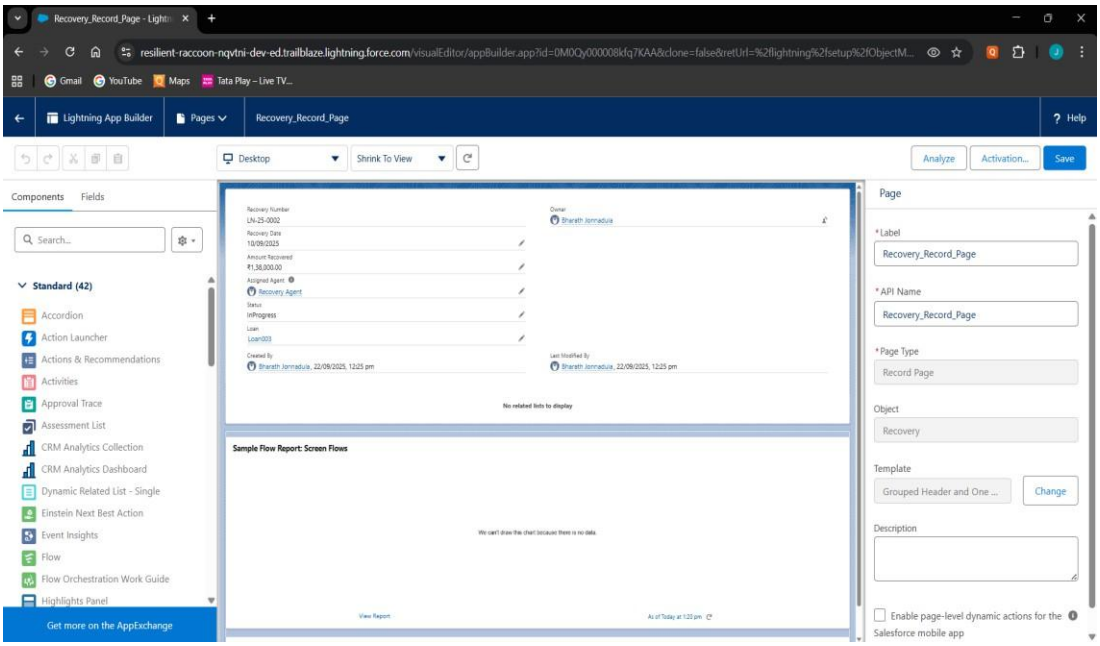


Record Pages

- Built custom Record Pages for Loan and Payment objects using Lightning App Builder.

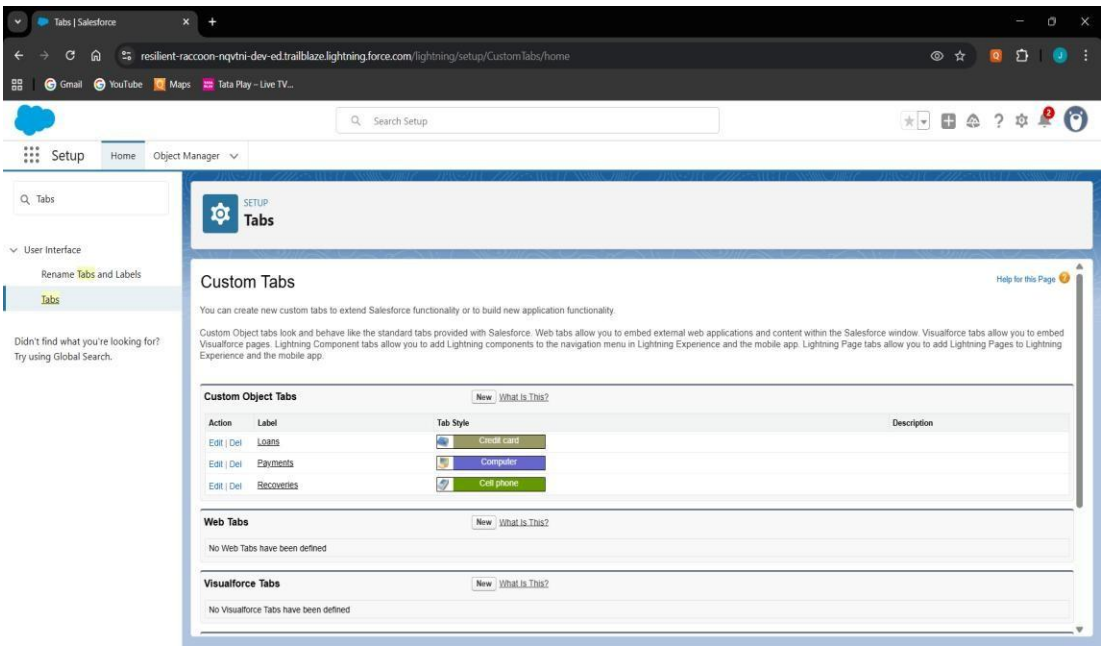






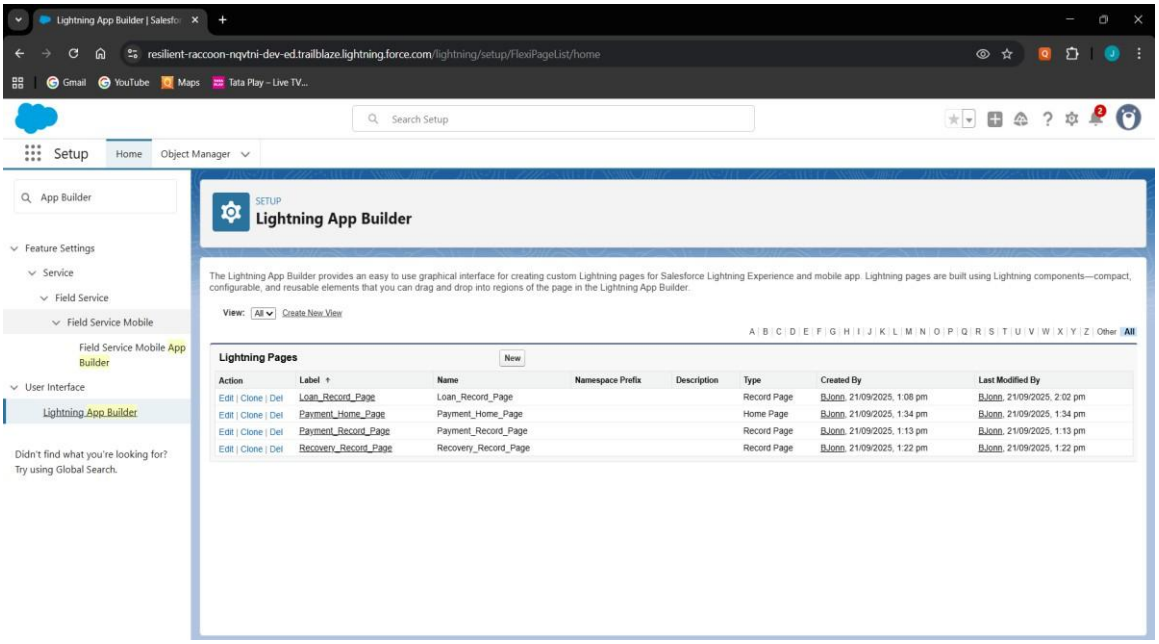
Tabs

- Added custom tabs for Loan, Payment, and Recovery objects.



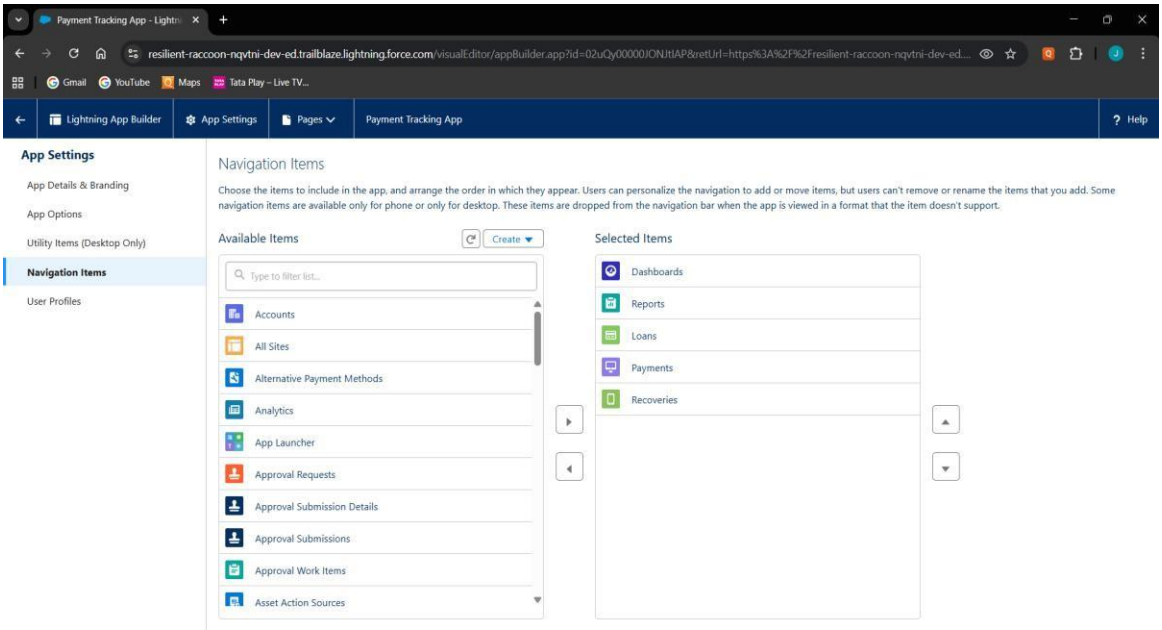
Home Page Layouts

- Customized the Home Page with components like Recent Loans, Pending Payments



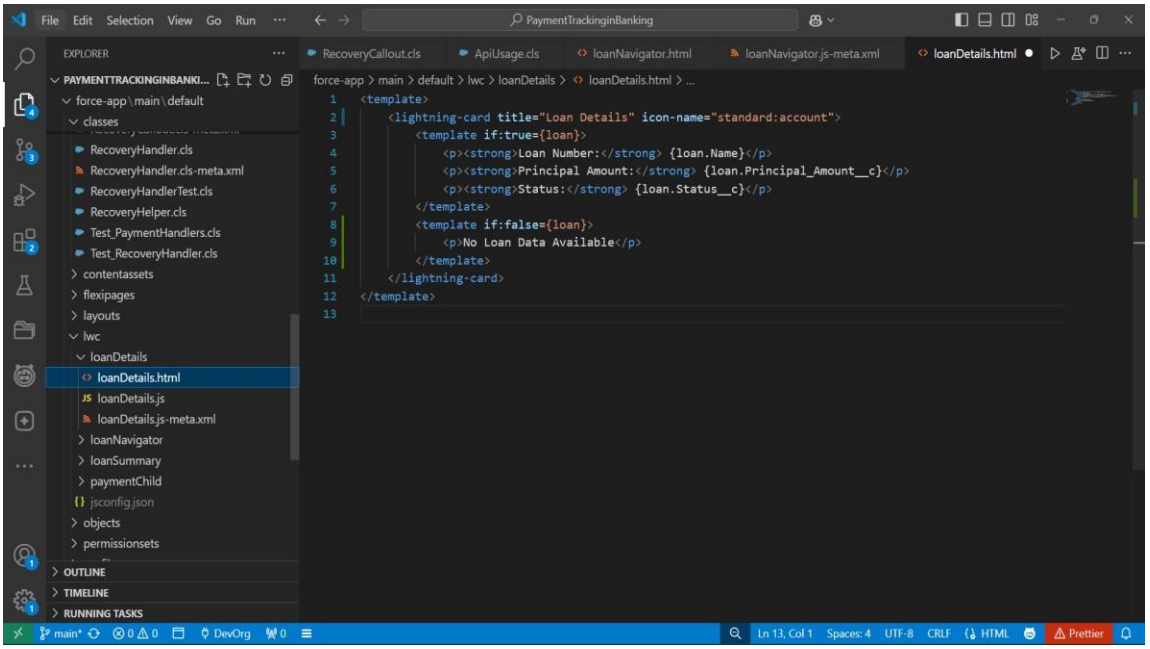
Utility Bar

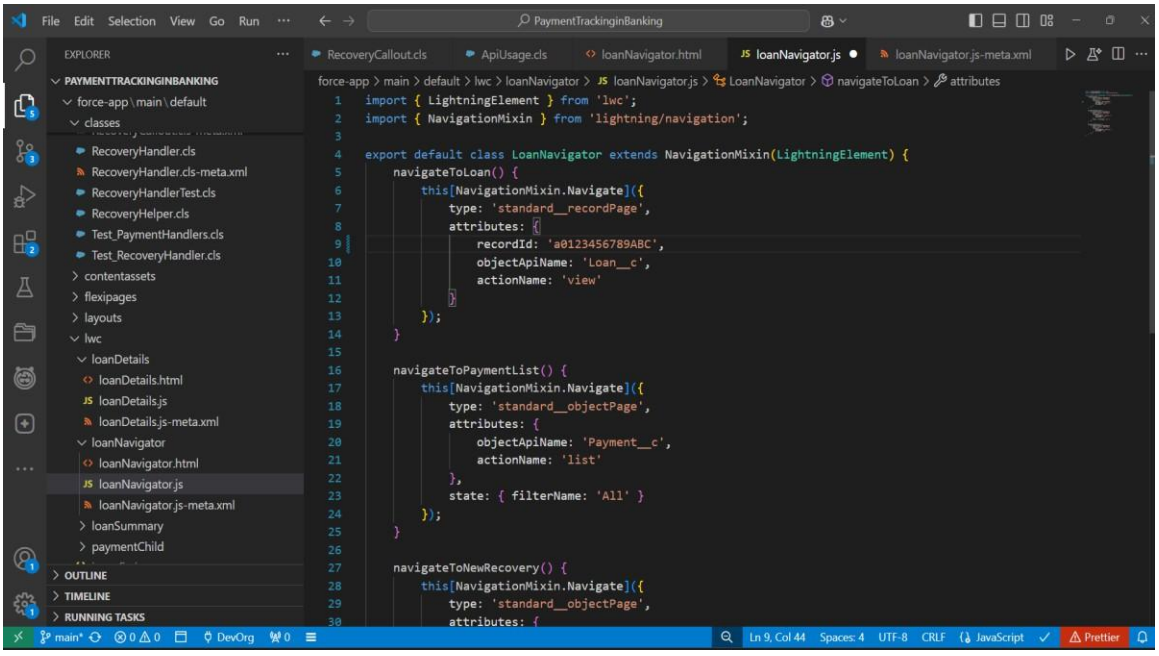
- Switch between loan & payment records.



LWC (Lightning Web Components)

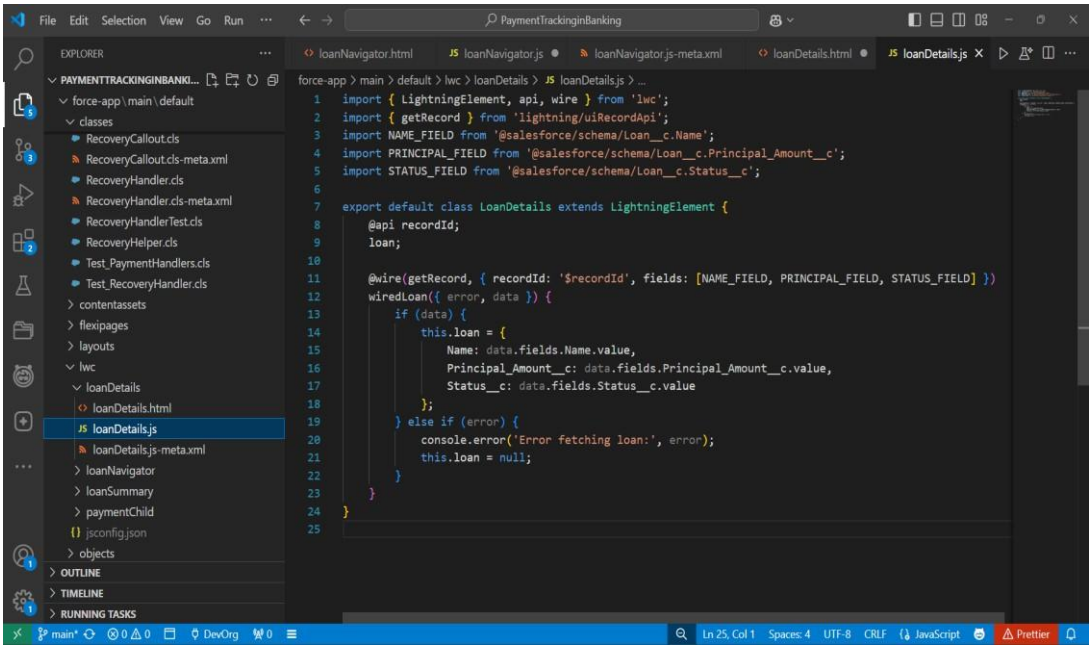
- Built LWC components (loanSummary, loanDetails, paymentChild).





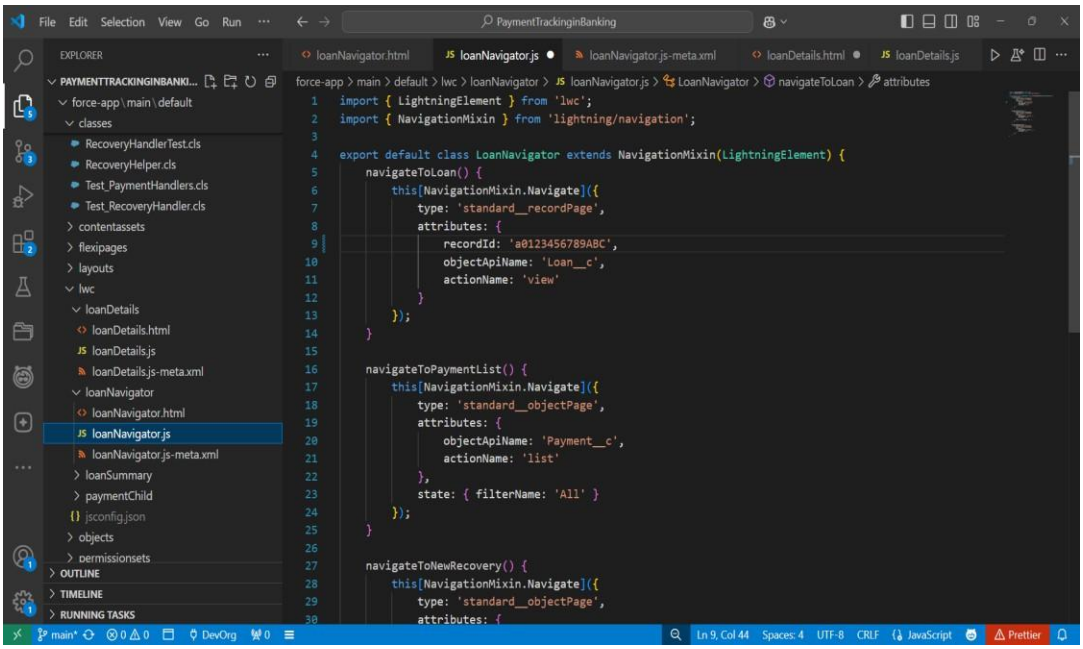
Wire Adapters

- Used @wire(getRecord) to fetch Loan fields directly in LWC.



Navigation Service

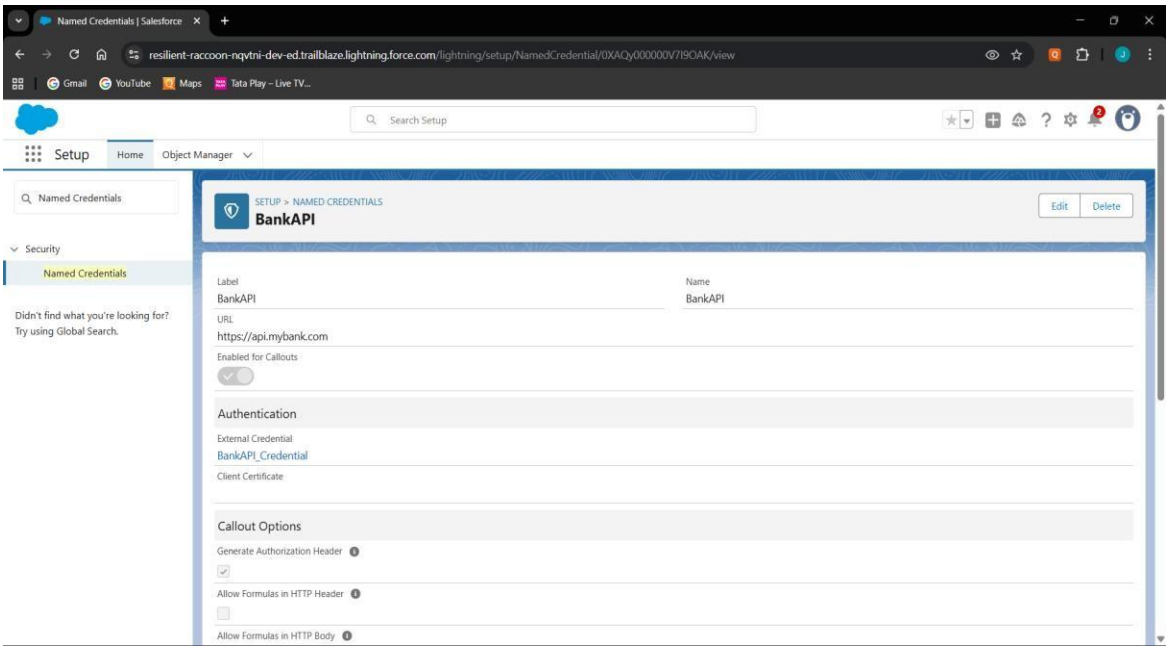
- Implemented NavigationMixin to let users navigate from Payment → Loan detail page.
- Improves user experience with one-click record redirection.



Phase 7: Integration & External Access

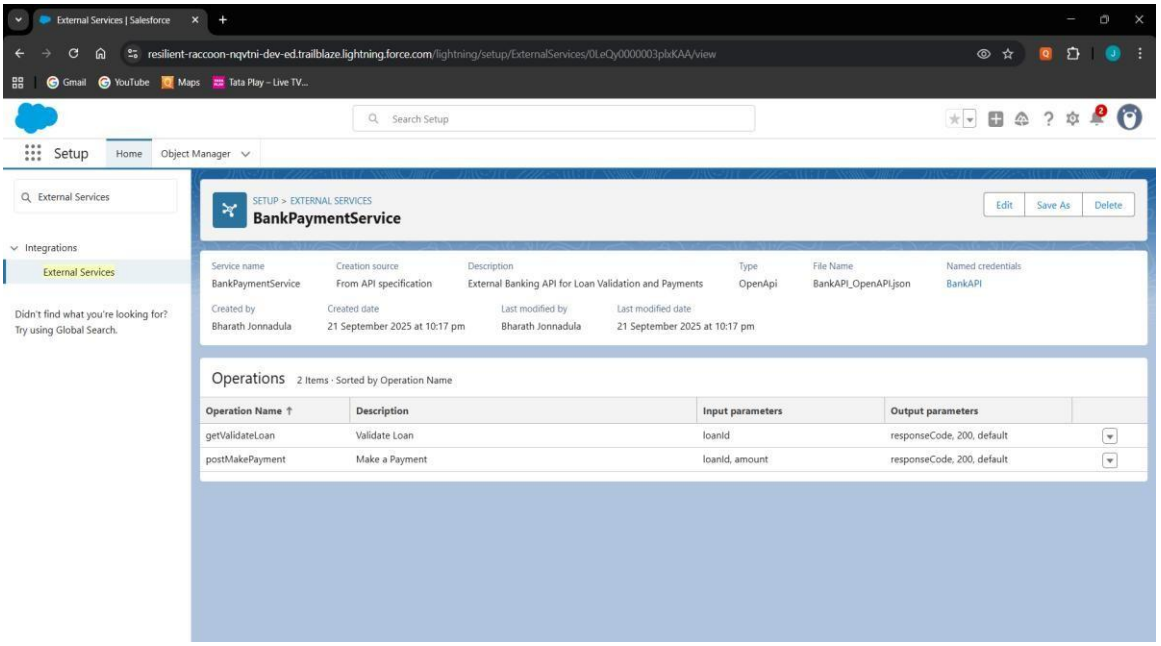
Named Credentials

- Used to securely store API endpoints + authentication details for external systems.



External Services

- Lets Salesforce call external APIs using schema (Swagger/OpenAPI).



Web Services (REST/SOAP)

- Allow external system to fetch Loan details.

LoanExternalService.cls

public with sharing class LoanExternalService {

public static String getLoanValidation(String loanId) {

HttpRequest req = new HttpRequest();

req.setEndpoint('callout:BankAPI/validateLoan?loanId=' + loanId);

req.setMethod('GET');

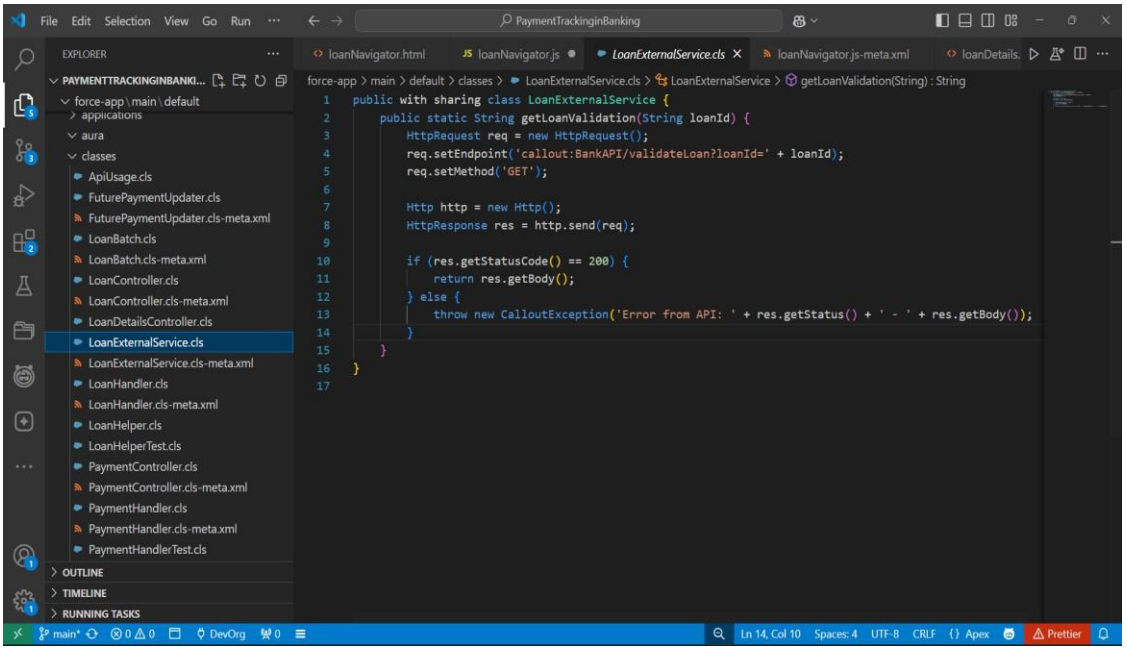
Http http = new Http();

HttpResponse res = http.send(req);

if (res.getStatusCode() == 200) {

return res.getBody();} else {

throw new CalloutException('Error from API: ' + res.getStatus() + ' - ' + res.getBody());}}}

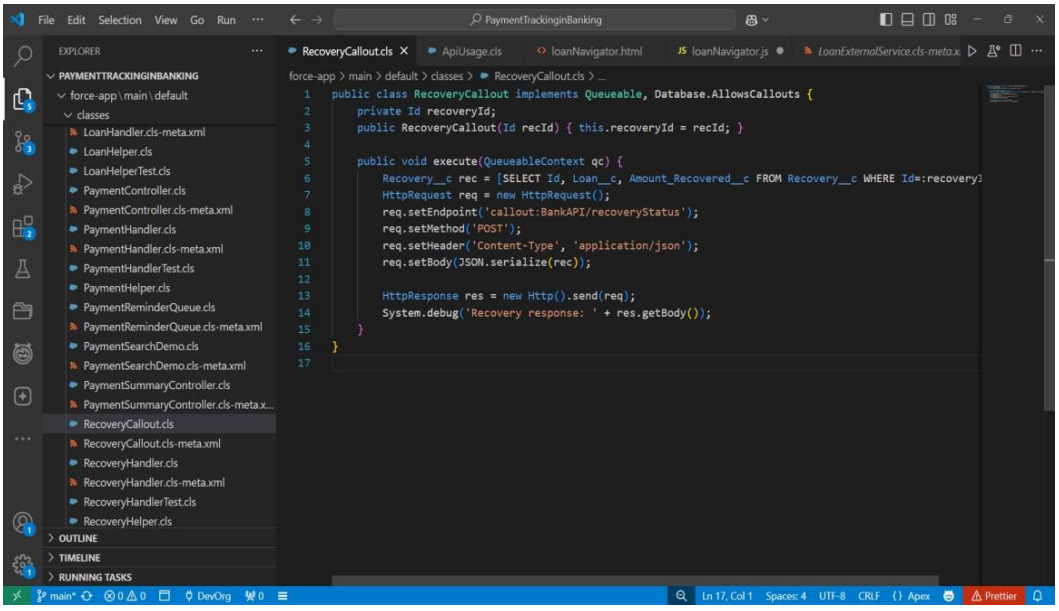


Callouts

- Used when Salesforce sends data to external services.

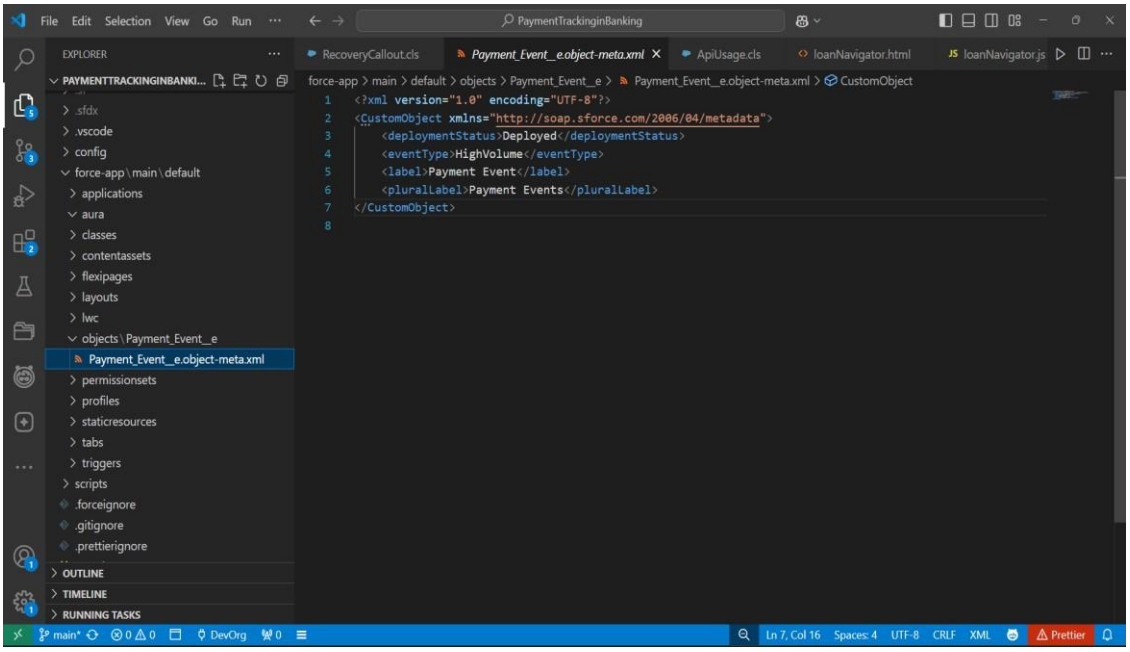
RecoveryCallout.cls

```
public class RecoveryCallout implements Queueable,
    Database.AllowsCallouts {

   private Id recoveryId;

   public RecoveryCallout(Id recId) { this.recoveryId = recId; }

   public void execute(QueueableContext qc) {

      Recovery__c rec = [SELECT Id, Loan__c, Amount_Recovered__c
   FROM Recovery__c WHERE Id=:recoveryId];

      HttpRequest req = new HttpRequest();

      req.setEndpoint('callout:BankAPI/recoveryStatus');

      req.setMethod('POST');

      req.setHeader('Content-Type', 'application/json');

      req.setBody(JSON.serialize(rec));

      HttpResponse res = new Http().send(req);

      System.debug('Recovery response: ' + res.getBody());}}
```



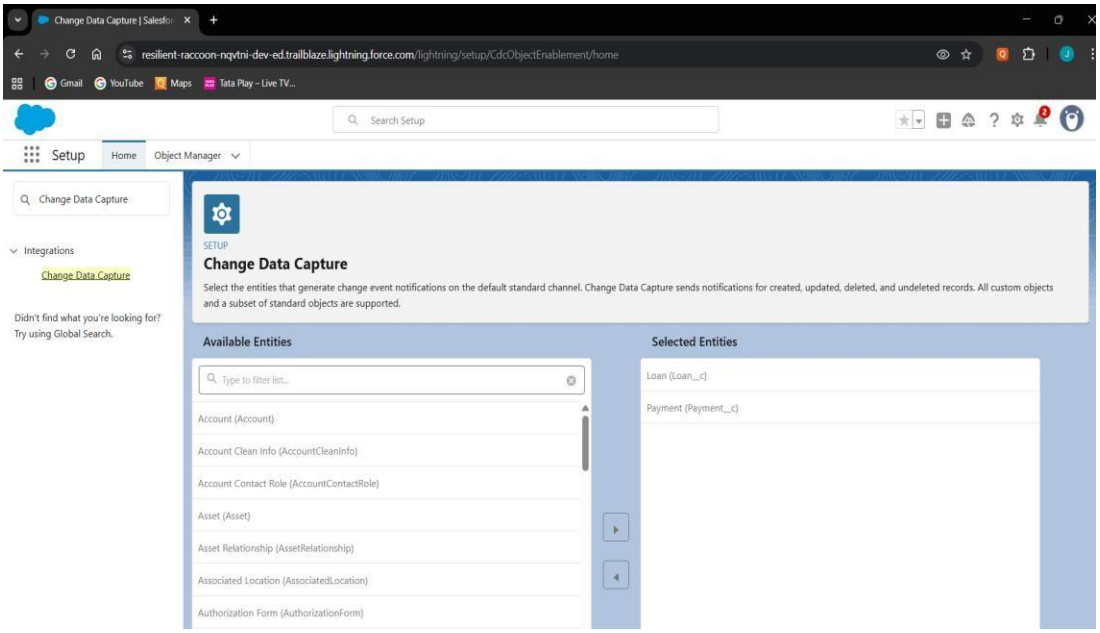
Platform Events

- Publish event when a Payment is Overdue (integrates with external collectors).



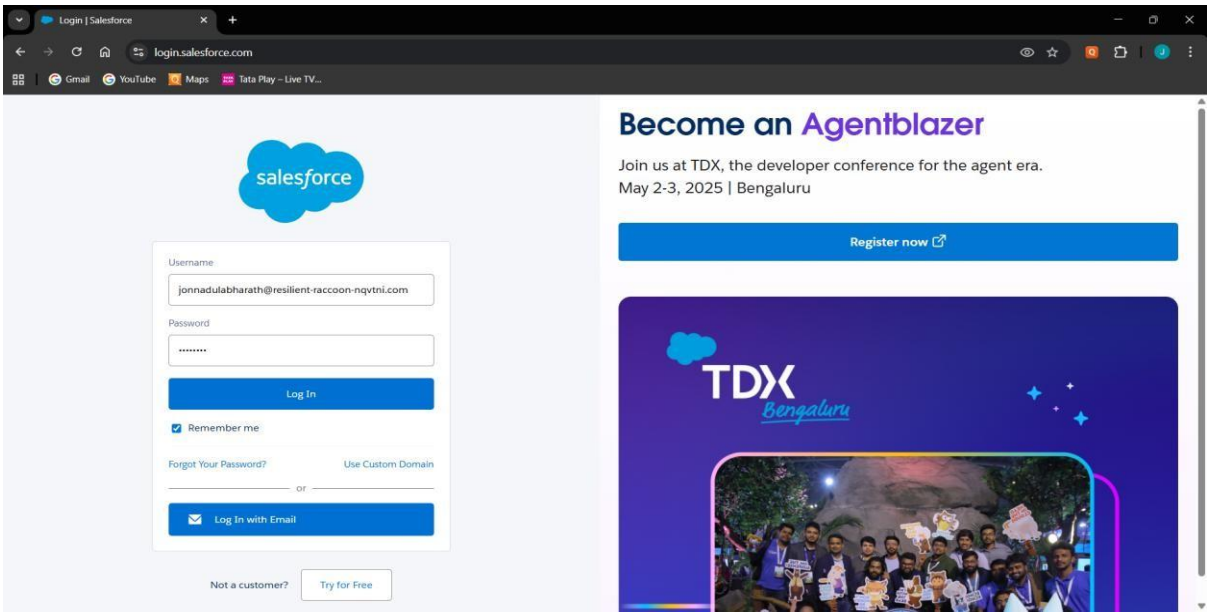
Change Data Capture (CDC)

- Streams real-time changes of Salesforce records to external systems.
- Any update in Payment_c record can be pushed automatically to reporting system.



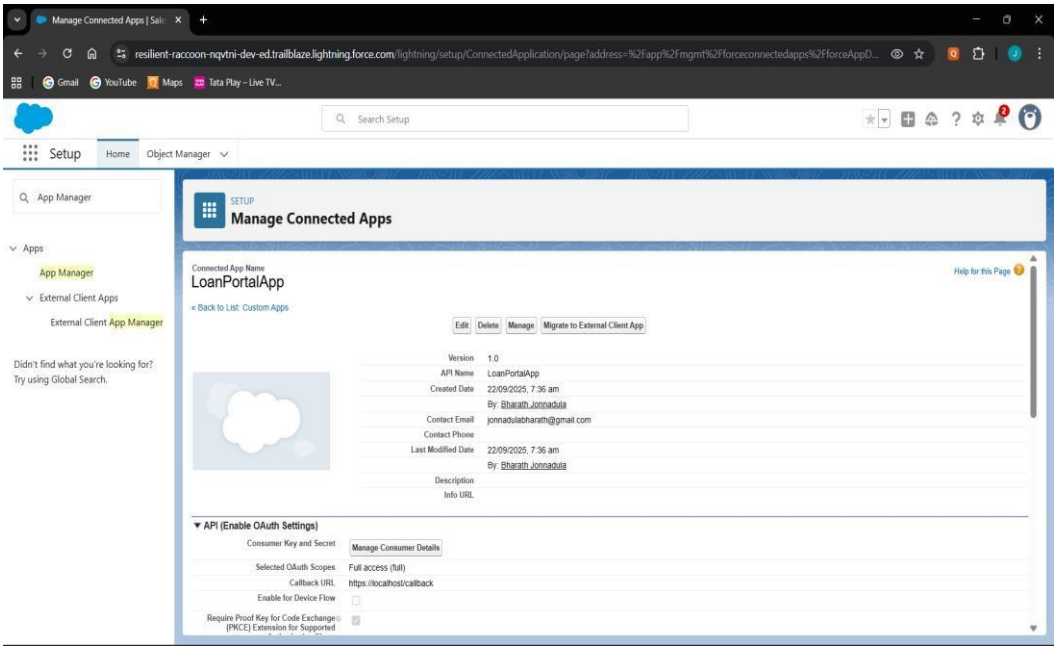
Salesforce Connect

- Provides real-time view of external data without storing in Salesforce.



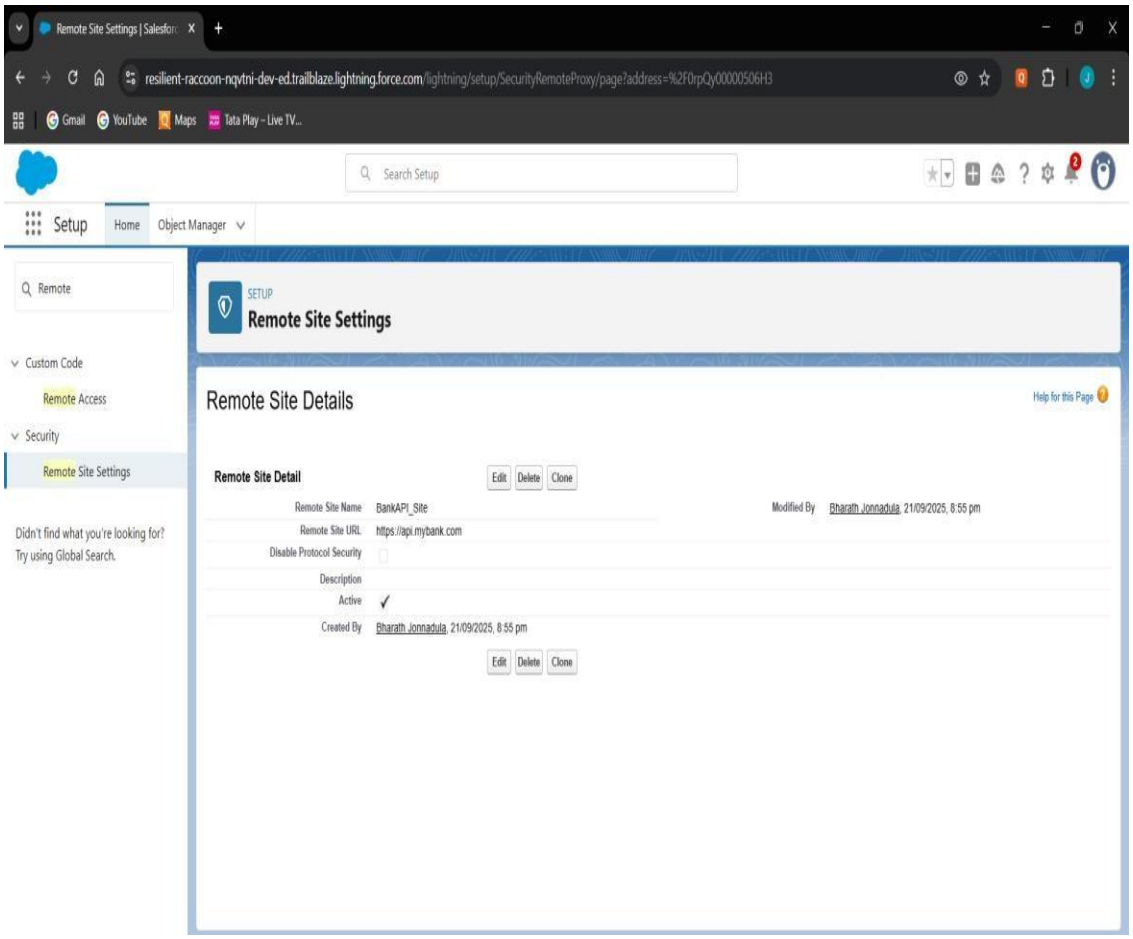
OAuth & Authentication

- Enables secure access for external apps to Salesforce APIs.



Remote Site Settings

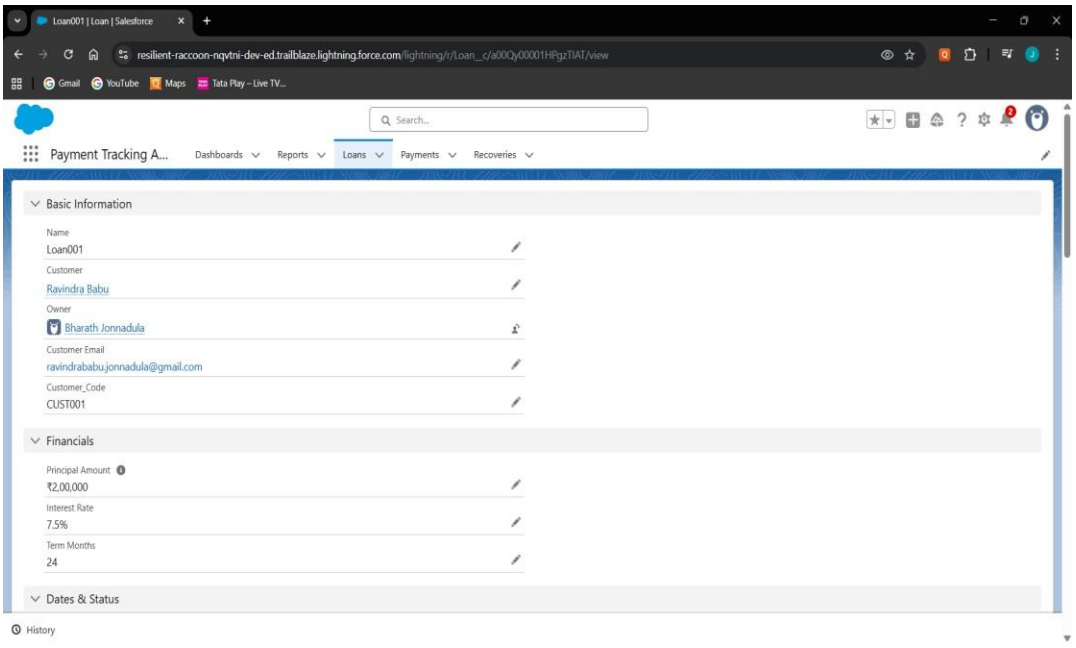
- Required to allow outbound callouts to untrusted domains.
- Add https://api.bank.com as a Remote Site Setting before making callouts.

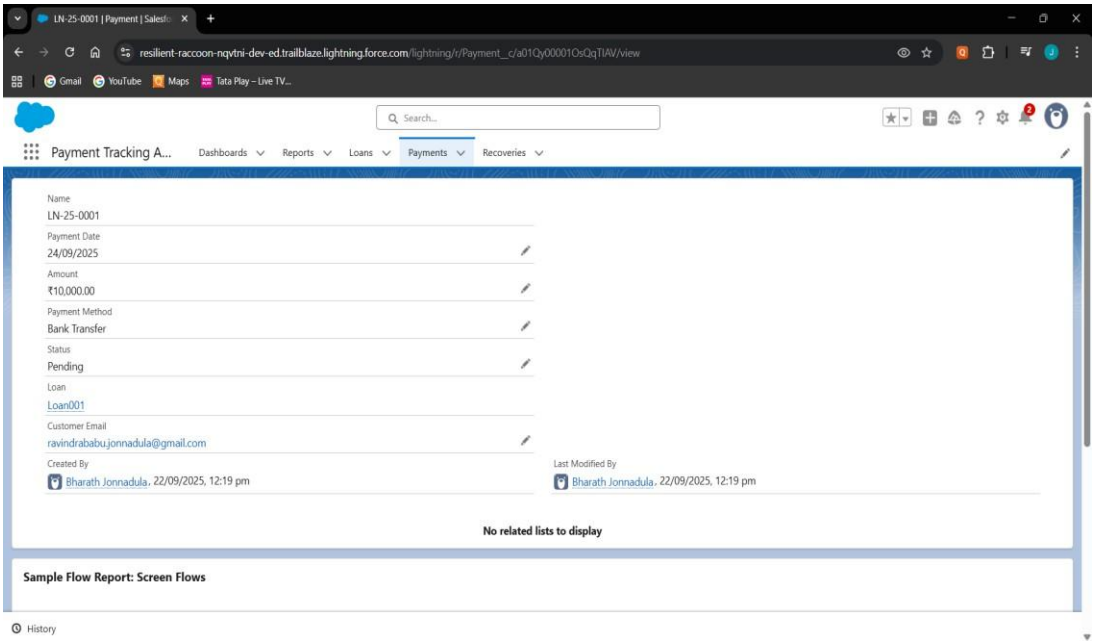


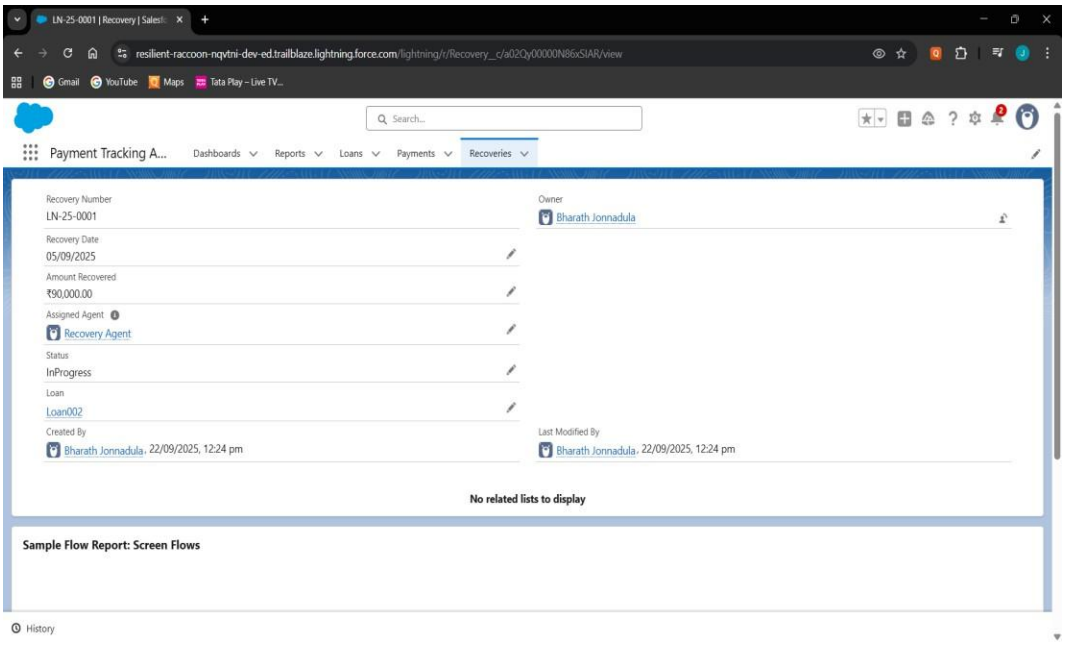
Phase 8: Data Management & Deployment

Data Import Wizard

- Used Data Import Wizard to insert sample data into Loan, Payment, and Recovery objects.





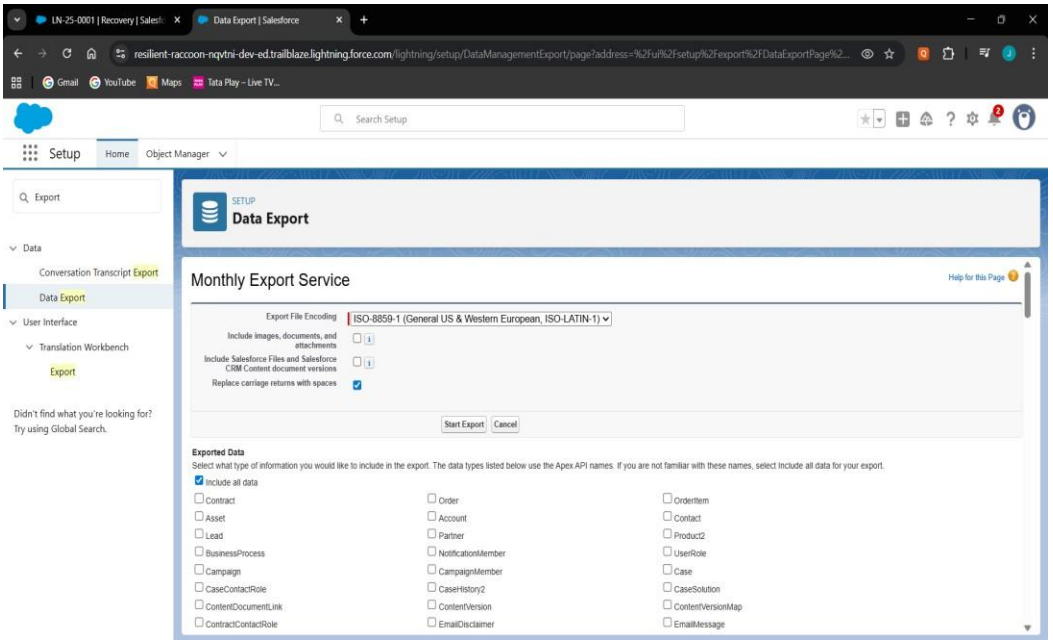


Data Loader

- Data Loader for bulk upload, but not required since records were limited.

Data Export & Backup

- Verified Data Export option in Setup for backup.



VS Code & SFDX

- Used VS Code with SFDX CLI for writing and deploying Apex Classes, Triggers, and LWC.
- Successfully pushed/pulled metadata between local project and Salesforce Org