Project Title: Gym Management System CRM

Submitted By: Tejasvini Sunil Chavhan
Prof. Ram Meghe Institute of Technology and Research, Badnera

Mandatory Sections to Include:

Problem Statement

In the fitness industry, gyms face multiple challenges in managing their members effectively. Traditional systems rely heavily on manual processes or fragmented tools that lead to inefficiencies such as:

- Difficulty in tracking member attendance, workout sessions, and renewals.
- Lack of personalized fitness and nutrition plans for members.
- Missed follow-ups with members who do not attend regularly or whose memberships are about to expire.
- Limited visibility into trainer performance and gym resource utilization.
- Inability to generate real-time dashboards and reports for business insights.

These issues result in **lower member satisfaction, reduced retention rates, and missed revenue opportunities**.
A **CRM-based Gym Management System** can solve these challenges by:

- Maintaining a centralized member database.
- Automating workout session scheduling and renewal reminders.
- Assigning trainers and tracking their performance.
- Creating customized fitness and diet plans.
- Providing dashboards and reports for decision-making.

Project Overview

The **Gym Management System CRM** is designed to streamline the operations of modern gyms and fitness centers by leveraging Salesforce CRM capabilities. The system will act as a **centralized platform** for managing members, trainers, workout sessions, fitness plans, and renewals, ensuring a smooth and engaging experience for both staff and members.

Who will use it?

- Trainers & Nutritionists → to track member treatment plans and follow-ups.
- Gym Admin Staff → to schedule workout sessions, manage records, and analyze reports.
- Members → to receive reminders, check workout sessions, and stay engaged with preventive care.

How it will help?
This solution ensures that members do not miss important medical workout sessions, improves preventive care participation, and reduces manual work for gym staff. It will provide real-time dashboards to management, helping them monitor member inflow, follow-up completion rates, and overall gym efficiency.

Objectives

The main goals of the project are:

1. Improve Efficiency – Automate workout session scheduling and follow-up reminders.
2. Automate Manual Tasks – Reduce manual tracking of member visits and preventive care programs.
3. Provide Better Reporting – Generate dashboards and reports on member inflow, missed vs. completed follow-ups, and preventive care participation.
4. Ensure Data Accuracy – Maintain complete member medical history and treatment details in one place.
5. Enhance Member Experience – Keep members engaged through timely notifications, reminders, and easy access to their workout session details.

Phase 1:- Problem Understanding & Industry Analysis

1. Requirement Gathering

- Collecting detailed requirements from gym owners, trainers, staff, and members.
- Key requirements include:
    - Member registration & profile management.
    - Subscription/plan management (monthly, quarterly, yearly).
    - Attendance tracking (biometric/RFID/manual).
    - Trainer allocation and schedule management.
    - Payment & billing integration.
    - Fitness progress tracking (BMI, workout logs, diet plans).
    - Notifications & reminders (SMS/Email/App).

2. Stakeholder Analysis

- **Primary Stakeholders:**
    - **Gym Members** → need seamless registration, tracking, and support.
    - **Trainers** → need access to schedules, member progress, and training plans.
    - **Gym Owners/Managers** → need complete control over finances, staff, and business insights.
- **Secondary Stakeholders:**
    - **Vendors/Equipment Suppliers** → integration for maintenance schedules.
    - **Third-party Payment Gateways** → secure transactions.

3. Business Process Mapping

- **Current Process (Manual):**
  Registration → Payment → Manual logbooks for attendance and schedules → Trainers manage progress offline → Owner manually checks revenue.
- **Proposed Process (With System):**
  Digital Registration → Automated Payment & Billing → Attendance via integrated system → Trainers update member progress online → Dashboard for owner to monitor revenue, staff performance, and member engagement.

4. Industry-specific Use Case Analysis

- **Fitness Industry Trends:**
    - Growing demand for digital gym management solutions.
    - Increased focus on personalized training and health tracking.

    - Integration with fitness wearables (Fitbit, Smartwatch, etc.).
    - Hybrid models (offline + online training).
- **Use Cases:**
    - **Member Management:** New member joins → Assigned trainer → Progress tracked digitally.
    - **Trainer Scheduling:** Automated shift management to avoid overlaps.
    - **Billing & Renewals:** Automatic reminders and invoices for expiring subscriptions.
    - **Analytics & Reporting:** Owners can view real-time business insights.

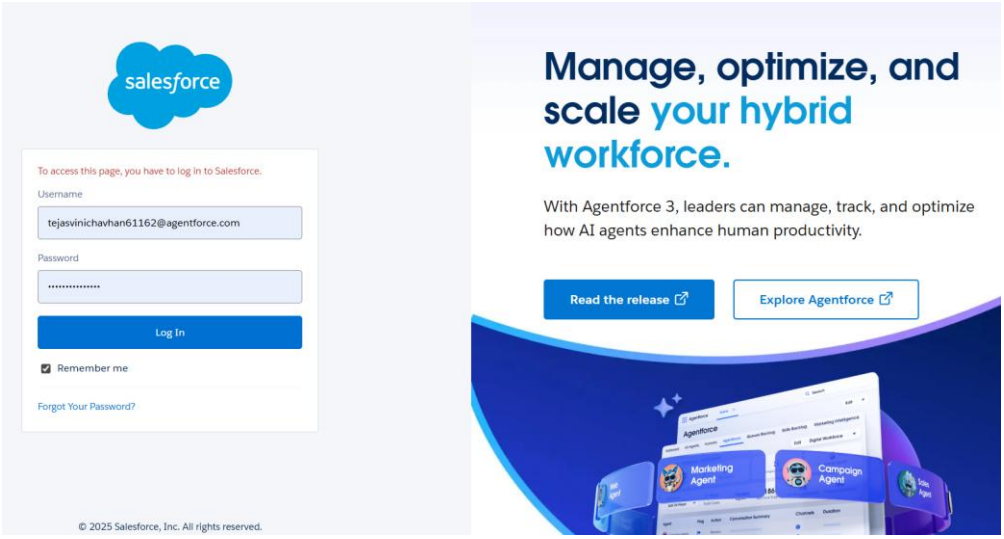
5. AppExchange Exploration

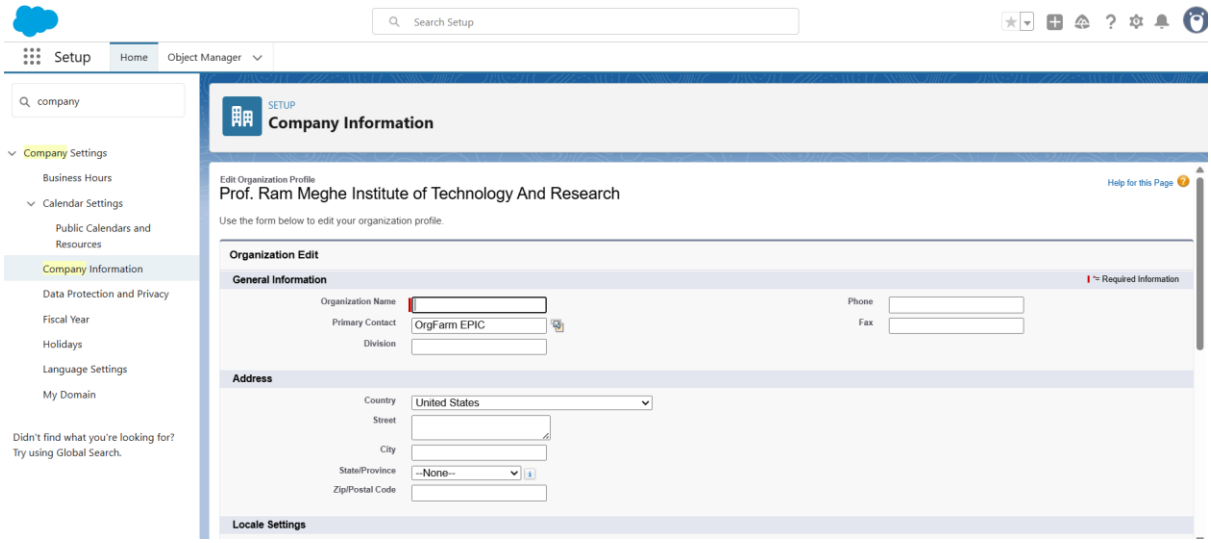
- Salesforce AppExchange offers pre-built apps and components that can be leveraged:
    - **Membership Management Apps** → Ready-to-use templates for managing members and subscriptions.
    - **Payment Integration Apps** → Connect to Stripe, PayPal, Razorpay for secure billing.
    - **Scheduling Tools** → Appointment booking and trainer allocation apps.
    - **Analytics & Dashboard Packages** → Customizable reports for business insights.
- Exploration of these apps can reduce development time and ensure industry best practices are incorporated.

Phase 2:- Org Setup & Configuration

1. Salesforce Edition & Dev Org Setup

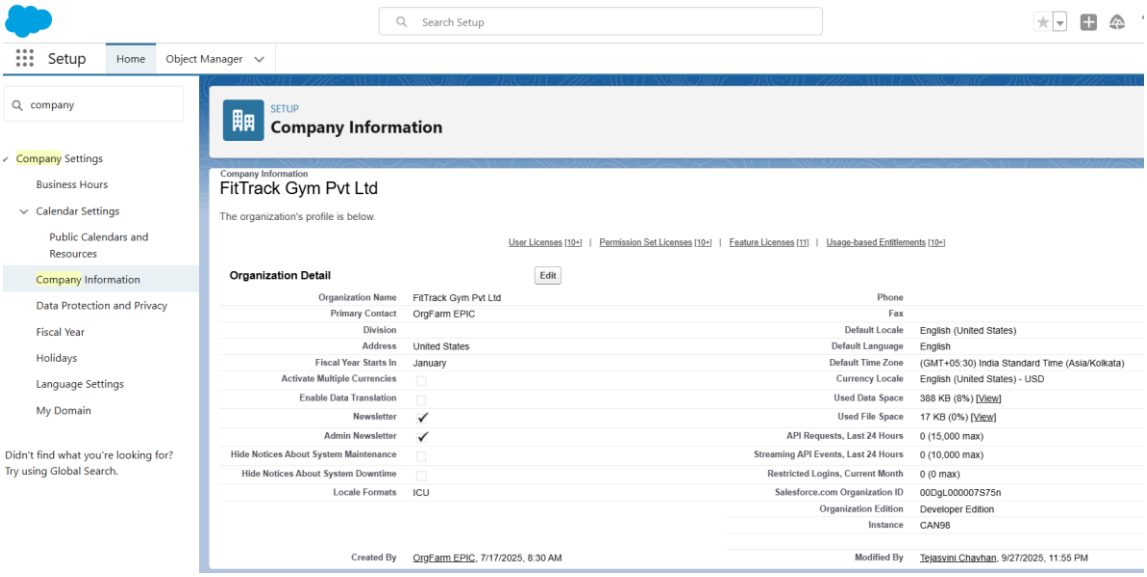
We used a **Salesforce Developer Edition Org** to build this project.





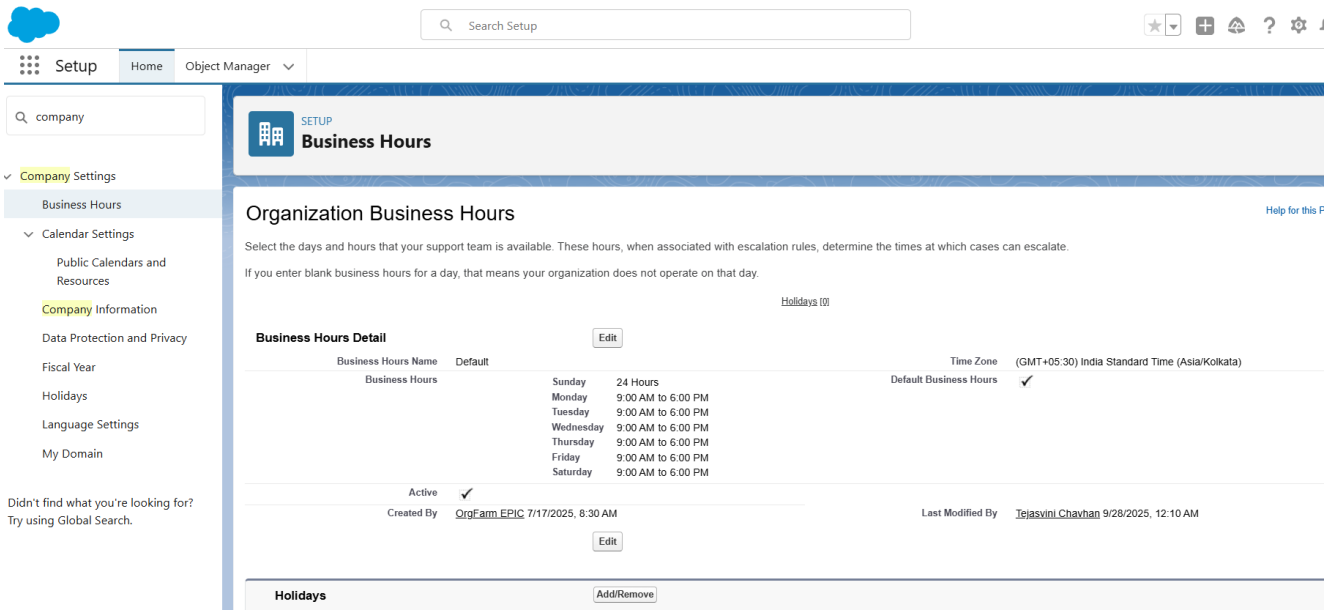
2. Company Profile Setup

- Company Name: FitTrack Gym Pvt Ltd
- Default Time Zone: *Asia/Kolkata (IST)*
- Fiscal Year: *Standard (Jan–Dec)*
- Default Locale: English (India)
- Default Language: English
- Default Currency: INR (Indian Rupee)

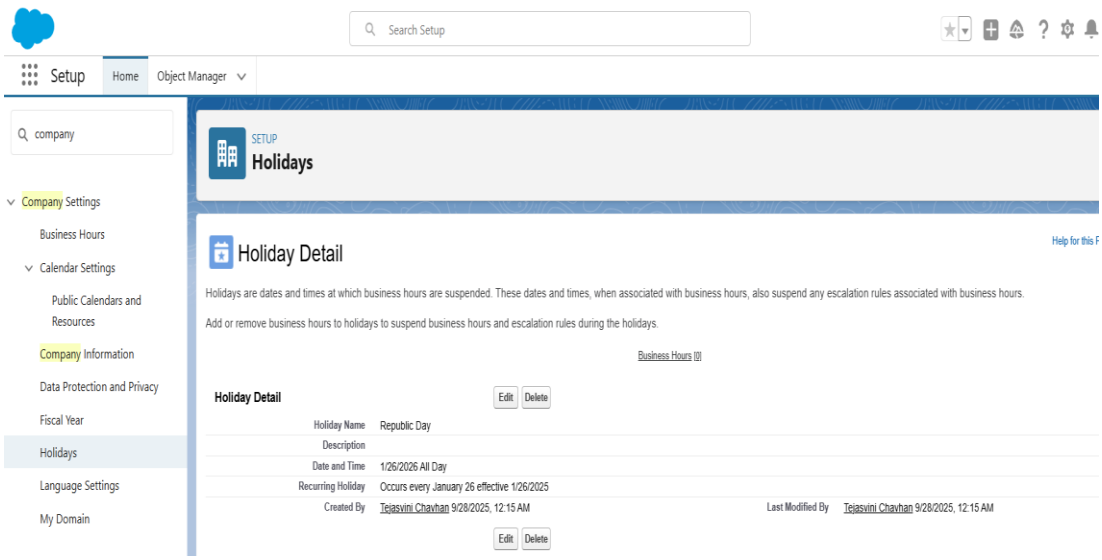


3.Business Hours & Holidays

9 AM – 6 PM IST, Mon–Sat

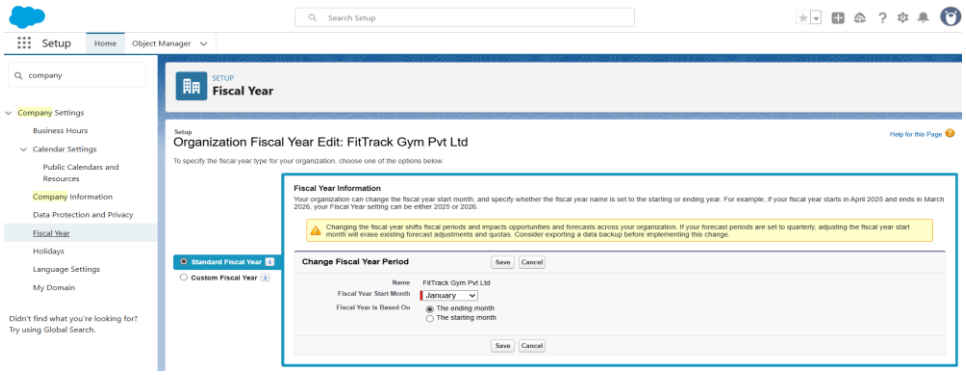


- Holidays Configured: *Republic Day, Independence Day*



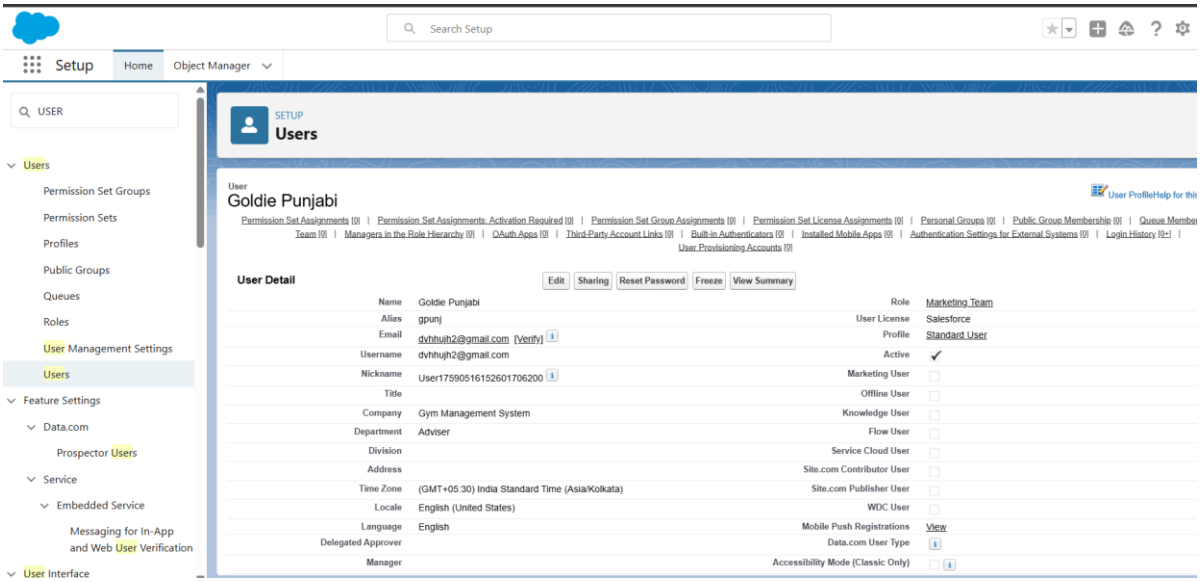
4. Fiscal Year Settings

- Fiscal Year: Standard (January-December).
- Custom Fiscal Year not enabled (not needed for this project).



5.User Setup & Licenses

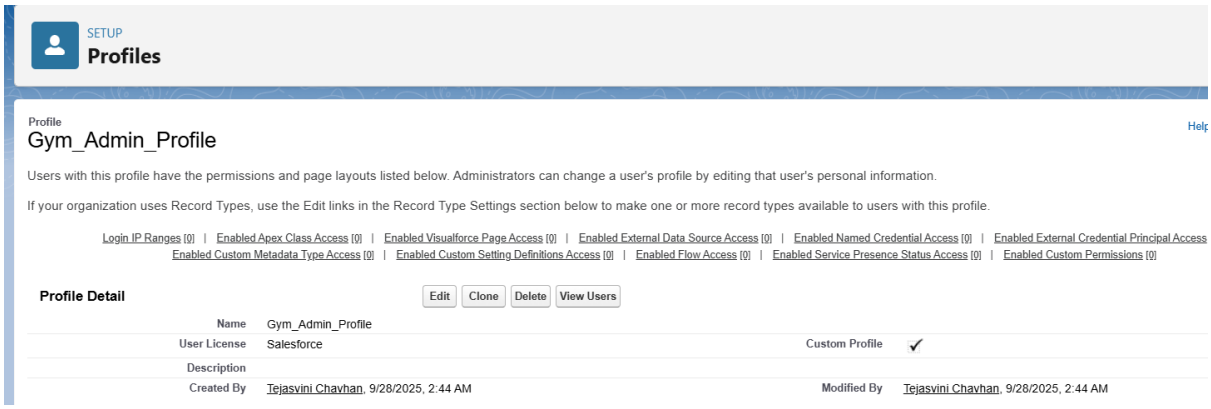
- Create sample users: Sales User, Business Manager, Support User.
- Assign Salesforce licenses.
- **Created Test User**
- First Name: Goldie
- Last Name: Punjabi
- Role: Marketing Team
- User License: Salesforce
- Profile: Standard User

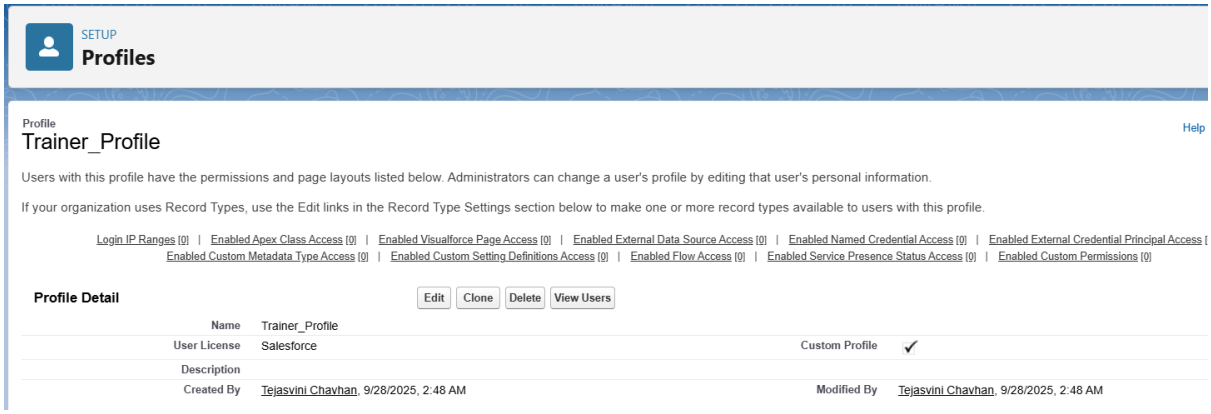


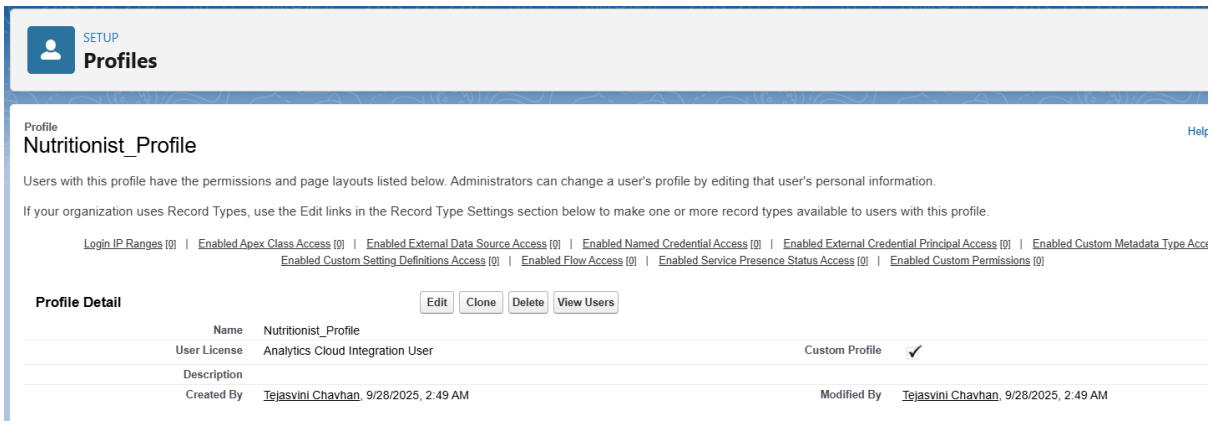
6. Profiles

Created/Cloned profiles to control object and field-level access.

- Gym_Admin_Profile → cloned from System Administrator.
- Trainer_Profile → cloned from Standard User.
- Nutritionist_Profile.
- Front Desk Staff_Profile.







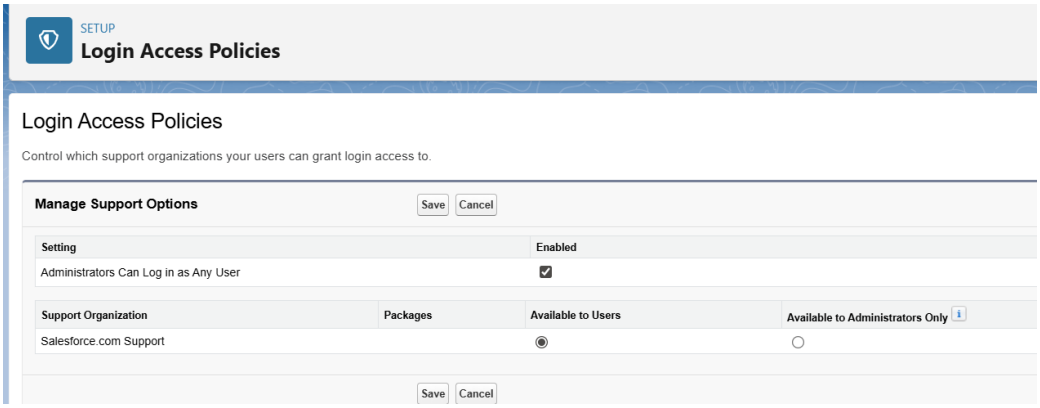
7.Roles

We created a role hierarchy to reflect gym structure:

CEO
↳ Gym Admin
  ↳ Gym Director
    ↳ Trainer
    ↳ Nutritionist
    ↳ Front Desk Staff

8. Login Access & Security

- Setup Login Access Policies
- Enable: Administrators Can Log in as Any User



9.Developer Org Setup

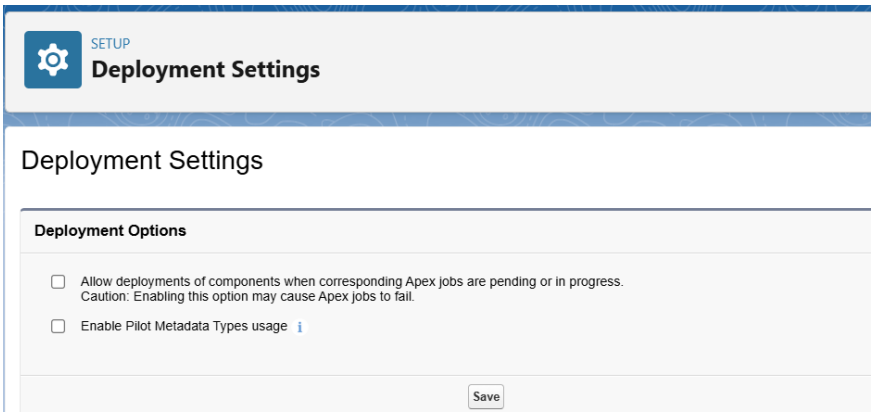
- Project developed in Salesforce Developer Edition Org.
- Provides free Enterprise-level features suitable for learning and development purposes.

10. Sandbox Usage

- Sandboxes are not available in Developer Edition.
- In real-world companies, sandboxes are used for testing, development, and training without affecting Production.

11. Deployment Basics

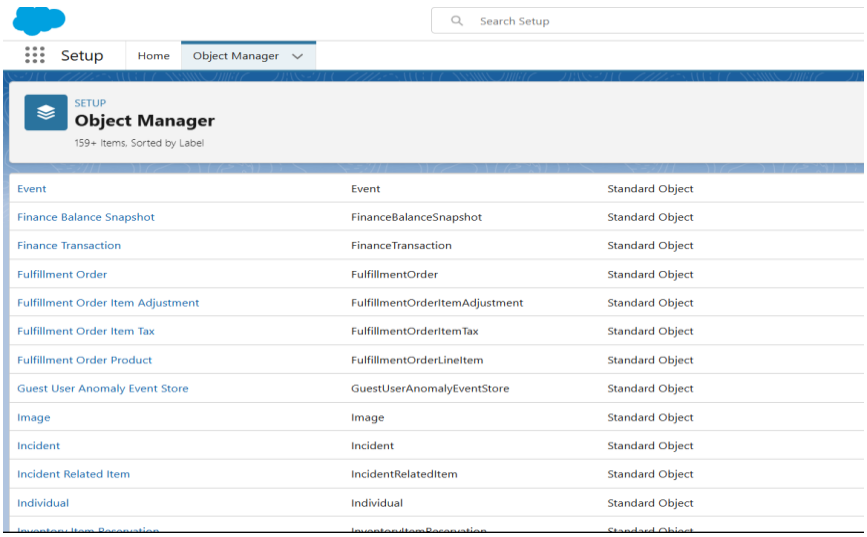
- Deployment methods include Change Sets or Salesforce CLI / VS Code.
- Not required here since the project is fully developed and tested within a single Developer Org.

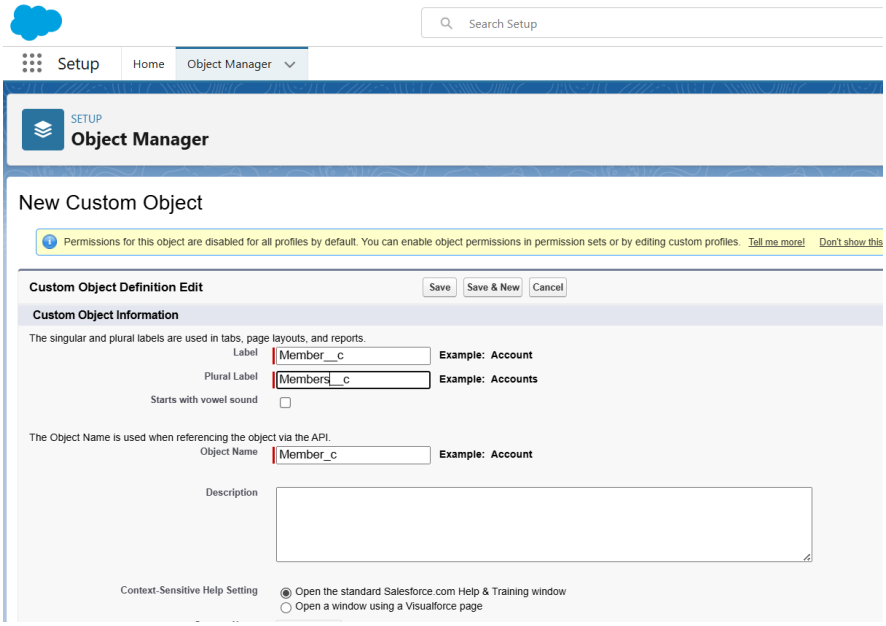


Phase 3 :- Data Modeling & Relationships

1. Standard & Custom Objects

- Used standard objects: User (Trainers, Nutritionists, Admins).
- Created custom objects:
    - Member__c (master record for members)
    - Workout_Session__c (linked to Member)
    - Fitness_Plan__c (linked to Member)
    - Membership_Renewal__c (linked to Member)
    - Clinic_Location__c (referenced from Workout_Session)

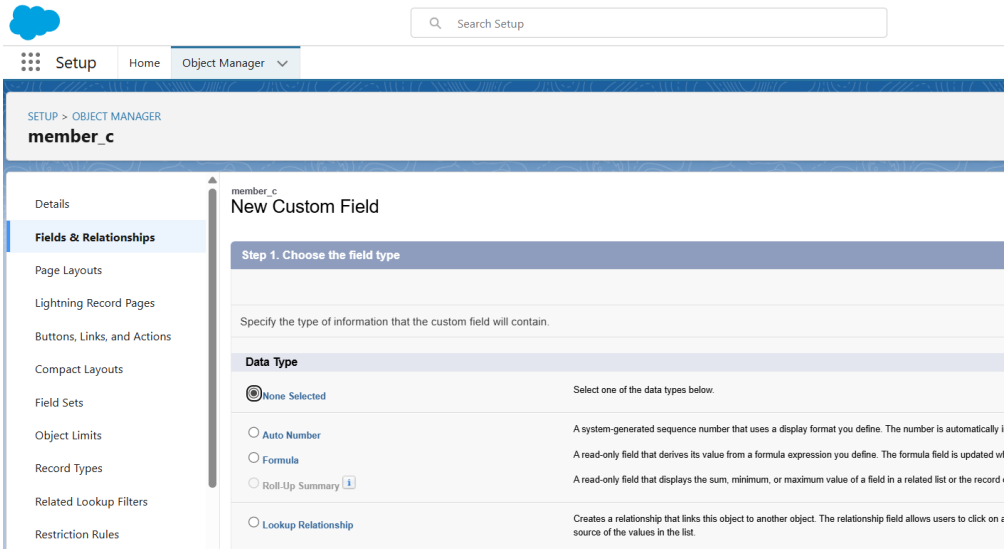




2. Fields

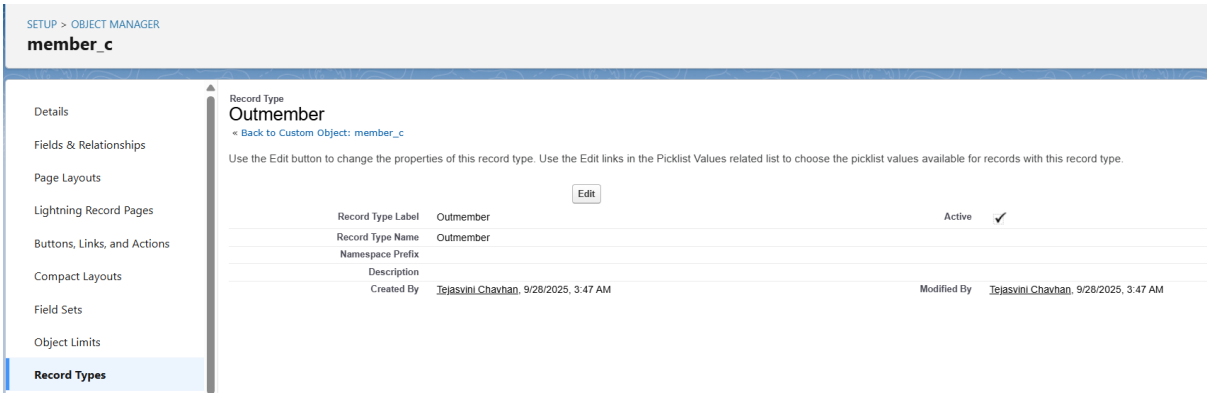
- Added key fields for each object:
    - Member__c → FirstName, LastName (required), Gender, DOB, Phone, Email, Assigned_Trainer__c (Lookup → User).

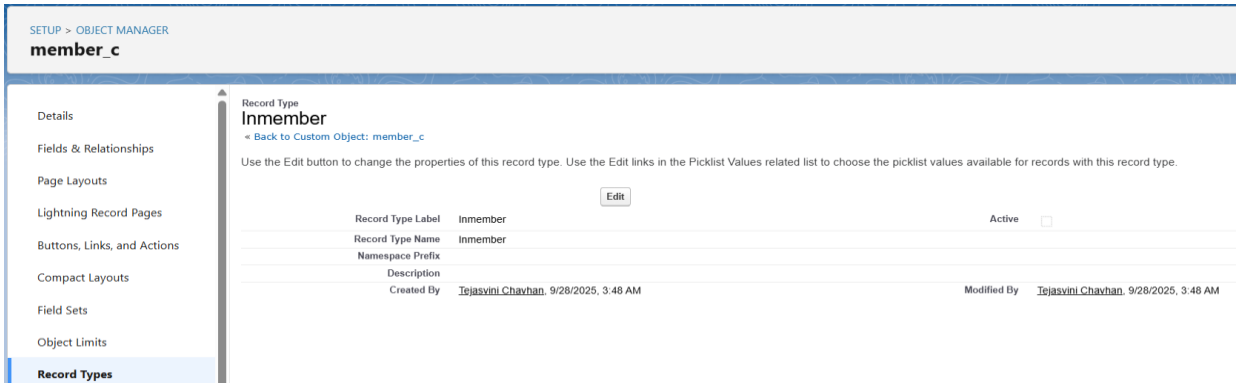
- Workout_Session__c → Workout_Session_Date__c, Status__c (Scheduled/Completed/Missed), Clinic_Location__c (Lookup → Clinic_Location__c).
- Fitness_Plan__c → Diagnosis__c, Medication__c, Next_Review_Date__c.
- Membership_Renewal__c → Renewal_Date__c, Notes__c.



3. Record Types

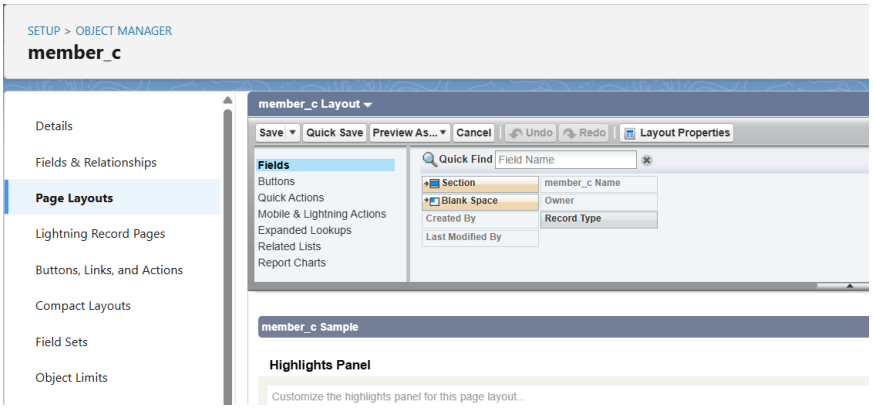
- Created Record Types for Member:
  - Outmember
  - Inmember
- Assigned Record Types to profiles:
  - Trainer_Profile → both Outmember & Inmember.
  - Front Desk Staff_Profile → Outmember only.





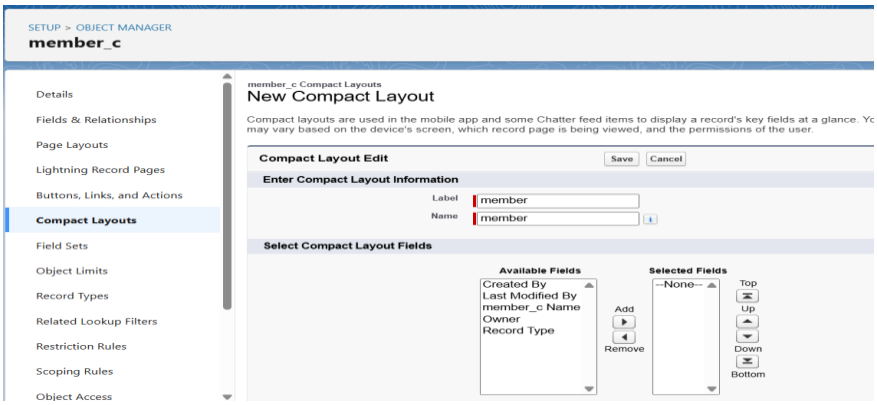
4. Page Layouts

- Created role-based layouts:
  - Trainer Layout → includes Medical Details, Workout_Sessions, Treatment Plans, Renewal Schedules.
  - Nutritionist Layout → shows Medical Details but hides sensitive free text fields.
  - Front Desk Staff Layout → only Personal Info + Workout_Sessions.



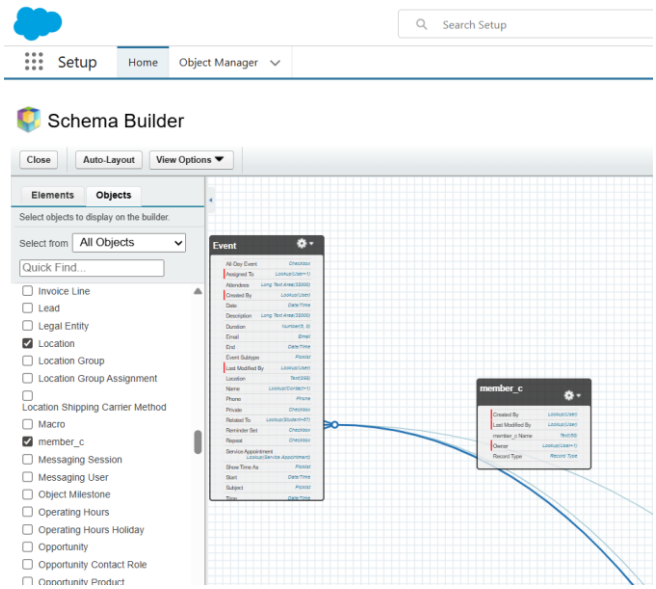
5. Compact Layouts

- Configured compact layout for Member:
  - Fields: LastName, Phone, Assigned_Trainer__c, Next_Renewal_Date__c.
- Compact layout set as **Primary** for all profiles.



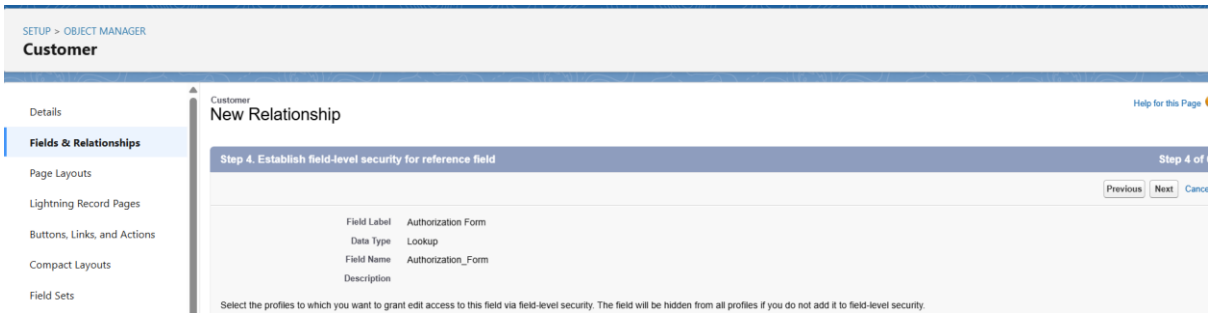
6. Schema Builder

- Used Schema Builder to visualize relationships:
  - Member__c (Master) → Workout_Session__c, Fitness_Plan__c, Membership_Renewal__c (Detail).
  - Workout_Session__c → Lookup → Clinic_Location__c.



7. Relationships (Lookup vs Master-Detail vs Hierarchical)

- Master-Detail: Member__c → Workout_Session__c, Fitness_Plan__c, Membership_Renewal__c.
- Lookup: Workout_Session__c → Clinic_Location__c.
- **Hierarchical:** Not used (only applies to User object).



8. Junction Objects

We created a **junction object** to handle many-to-many relationships.

Use Case in Healthcare CRM:

- A member can be treated by multiple trainers (specialists).
- A trainer can treat multiple members.
  👉 To model this, we created a Member_Trainer_Assignment__c junction object.

Configuration:

- Custom Object: Member_Trainer_Assignment__c
- Fields:
  - Member__c (Master-Detail → Member)
  - Trainer__c (Master-Detail → User)

Result:

- Many-to-Many between Member and Trainer via Member_Trainer_Assignment.
- Used in reports/dashboards to see "which trainer is treating which members."

9. External Objects

We configured an External Data Source to prepare for integration with gym billing/lab systems.

Steps:

1. Created Named Credential: GymBilling_NC (No Authentication, placeholder).
2. Created External Data Source: GymBilling_EDS (OData 4.0).
3. Attempted Validate & Sync → error due to Developer Org proxy.

Result:

- External Objects (Invoice__x, Customer__x) were planned but not synced in Dev Org.
- This step will be completed in **Phase 7 (Integrations)**.

Phase 4 — Process Automation (Admin)

Goal: Automate validation, reminders, approvals and routine actions so members don't miss follow-ups and staff get notified automatically.

1. Validation Rules

Purpose: To enforce data quality by preventing invalid records.
Example: Prevent users from creating workout sessions in the past.

Steps:

1. Setup → Object Manager → Workout_Session → Validation Rules → New.
2. Rule Name: Prevent_Workout_Session_In_Past.
3. Formula:
4. Workout_Session_Start__c < NOW()
5. Error Message: Workout_Session cannot be scheduled in the past.

2. Workflow Rules (Legacy)

Purpose: Legacy automation (not recommended for new builds, but demonstrated).
Example: When an workout session is marked "Missed", create a follow-up Task.

Steps:

1. Setup → Workflow Rules → New Rule → Object = Workout_Session.
2. Criteria: Workout_Session_Status__c = 'Missed'.
3. Immediate Action → New Task.
    - Subject: Follow up missed workout session
    - Assigned To: User/Queue
    - Priority: High
4. Save & Activate.

3.Process Builder (Legacy)

Purpose: Automate creation of follow-up schedule for members with chronic disease.

Steps:

1. Setup → Process Builder → New.
2. Object: Member, Start when record is created/edited.
3. Criteria: Chronic_Disease__c = TRUE.
4. Immediate Action → Create Record → Membership_Renewal__c.
    - Member__c = [Member].Id
    - Scheduled_Date__c = Today + 30.
5. Save & Activate.

4. Approval Process

Purpose: Require approval before cancelling an workout session.

Steps:

1. Setup → Approval Processes → Object = Workout_Session.
2. Name: Workout_Session_Cancel_Approval.
3. Entry Criteria: Cancel_Request__c = TRUE.
4. Approver: Gym Admin (or assigned role).
5. Final Approval Action:
    - Field Update → Workout_Session_Status__c = Cancelled.
    - Email Alert → Member.
6. Final Rejection Action: Reset Cancel_Request__c = FALSE.
7. Add Submit for Approval to Workout_Session Page Layout.
8. Save & Activate.

5. Flow Builder

Flows are the recommended automation tool. We built **four types** of flows.

A) Record-Triggered Flow — Workout_Session Reminder (24 hours before)

1. Flow → New → Record-Triggered Flow.
2. Object = Workout_Session.
3. Entry Conditions:
    - Workout_Session_Status__c = Scheduled
    - Workout_Session_Start__c != null
4. Add Scheduled Path → Offset = -1 Day before Workout_Session_Start__c.
5. Action: Send Email Alert (Workout_Session_Reminder_Template).
6. Save & Activate.

B) Schedule-Triggered Flow — Daily Batch Reminders

1. Flow → New → Schedule-Triggered Flow.
2. Schedule: Daily at 6:00 AM.
3. Get Records: Workout_Sessions where Workout_Session_Start__c = Tomorrow and Status = Scheduled.
4. Loop through records.
5. Inside loop: Send Email / Create Task / Custom Notification.
6. Save & Activate.

C) Screen Flow — Front Desk Staff Quick Schedule

1. Flow → New → Screen Flow.
2. Add Screen:
    - Record Choice Set → Member.
    - Date/Time Picker → Workout_Session Date.
    - Text → Reason.
3. Create Records: Workout_Session.
    - Member__c = {!MemberChoice}
    - Workout_Session_Start__c = {!apptDateTime}
    - Status = Scheduled.
4. Save & Activate.
5. Add as Quick Action to Member record page.

D) Auto-Launched Flow — Reusable Logic

1. Flow → New → Auto-Launched Flow.
2. Variable: memberId (Text, Available for Input).
3. Formula: Follow_Up_Date = TODAY() + 30.
4. Create Records: Membership_Renewal__c.
    - Member__c = {!memberId}
    - Follow_Up_Date__c = {!Follow_Up_Date}.
5. Save & Activate.

6. Email Alerts

Purpose: To notify members of upcoming workout sessions.

Steps:

1. Setup → Email Templates → New → Lightning Email Template.
    - Related Entity = Workout_Session.
    - Subject: Reminder: Workout_Session on {!Workout_Session.Workout_Session_Start__c}.
    - Body: Hello {!Workout_Session.Member__r.Name}, your workout session is scheduled for {!Workout_Session.Workout_Session_Start__c}.
2. Setup → Email Alerts → New.
    - Object: Workout_Session.
    - Email Template: Workout_Session_Reminder_Template.
    - Recipient: Member.Email.
3. Save.

7. Field Updates

Purpose: Mark workout sessions as completed when follow-up is done.

Steps:

1. Flow → New → Record-Triggered Flow.
2. Object: Workout_Session.
3. Entry Condition: Renewal_Completed__c = TRUE.
4. Action: Update Records (or Assignment in before-save).
    - Set Workout_Session_Status__c = Completed.
5. Save & Activate.

8. Tasks

Purpose: Create Tasks when workout sessions are missed.

Steps:

1. Flow → New → Record-Triggered Flow.
2. Object: Workout_Session.
3. Condition: Workout_Session_Status__c = Missed.
4. Action: Create Task.
    - Subject: Follow up missed workout session for {!$Record.Member__r.Name}.
    - Priority: High.
    - Status: Not Started.
    - Due Date: Today + 1.
    - WhatId = Member__c.
    - OwnerId = Front Desk Staff/Owner.
5. Save & Activate.

9. Custom Notifications

Purpose: Send in-app/mobile reminders.

Steps:

1. Setup → Notification Builder → Custom Notifications → New.
    - Name: Workout_Session_Reminder.
    - Channels: Desktop & Mobile.
2. Flow Action → Send Custom Notification.
    - Type: Workout_Session_Reminder.
    - Title: Workout_Session Reminder.
    - Body: Workout_Session tomorrow at {!$Record.Workout_Session_Start__c}.
    - Recipient: User Id (admin or assigned).
    - Target Record Id = Workout_Session Id.
3. Save & Activate.

Phase 5 – Apex Programming (Developer)

Goal:-

To extend the Gym Management System CRM with programmatic automation using Apex. This phase demonstrates how to handle complex business logic (missed workout sessions, follow-ups, reminders) where declarative tools (Flows/Process Builder) may not be sufficient.

1. Apex Class

Use Case

We created an Apex Class MemberRenewalService to encapsulate reusable logic for creating Follow-Up Schedule records. Instead of writing the same logic in multiple places, this class centralizes the functionality. Trainers or staff may use this service (indirectly via triggers or scheduled jobs) to automatically generate follow-up reminders for members.

MemberRenewalService.apxc:-

```
public with sharing class MemberRenewalService {

    public static void createRenewal(Id memberId, Date followUpDate) {

        Membership_Renewal__c f = new Membership_Renewal__c(

            Member__c = memberId,

            Follow_Up_Date__c = followUpDate,

            Follow_Up_Notes__c = 'Auto-created via Apex Service'

        );
```

```
  insert f;

    }

}
```

2. Apex Trigger

Use Case

We implemented a trigger Workout_SessionTrigger on the Workout_Session__c object. The business need was:

- When an Workout_Session is marked as Missed, the system should automatically create a Follow-Up Schedule record.
- The follow-up is set 7 days after the missed workout session, and a note is added.

This ensures that no missed workout session is forgotten, improving member engagement and continuity of care.

Workout_SessionTrigger.apxt:-

```
trigger Workout_SessionTrigger on Workout_Session__c (after update) {

    List<Membership_Renewal__c> newRenewals = new List<Membership_Renewal__c>();



    for (Workout_Session__c appt : Trigger.new) {

        Workout_Session__c oldAppt = Trigger.oldMap.get(appt.Id);



        if (appt.Status__c == 'Missed' && oldAppt.Status__c != 'Missed') {

            newRenewals.add(new Membership_Renewal__c(

                Member__c = appt.Member__c,

                Follow_Up_Date__c = Date.today().addDays(7), // ✅ corrected

                Follow_Up_Notes__c = 'Auto-created for missed workout session'

            ));

        }

    }
```

```
    if (!newRenewals.isEmpty()) {
        try {
            insert newRenewals;
        } catch (DmlException e) {
            System.debug('Error while inserting follow-ups: ' + e.getMessage());
        }
    }
}
```

3. SOQL Queries

Use Case

We used SOQL (Salesforce Object Query Language) to fetch members with upcoming follow-ups in the next 7 days. This allows gym staff or administrators to generate reports and track which members need to be contacted.

QUERY:-

```
SELECT Id, Name,Phone_Number__c
FROM Member__c
WHERE Id IN (
    SELECT Member__c
    FROM Membership_Renewal__c
    WHERE Follow_Up_Date__c = NEXT_N_DAYS:7
)
```

4. Scheduled Apex

Use Case

We developed DailyRenewalScheduler, a scheduled Apex class that runs every morning at 7 AM. Its purpose is to find all members who have follow-ups due today and send reminders (or log notifications for now).

This ensures gym staff are aware of daily follow-ups without manually running reports.

5. Exception Handling

Use Case

In our trigger, we wrapped the insert operation in a try-catch block. This ensures that even if one record fails (e.g., due to validation issues), the system logs the error instead of breaking the entire transaction.This adds robustness to the automation and helps admins debug issues without impacting users.

```
try {
    // Missing required Member__c field → will cause DML error
    Membership_Renewal__c f = new Membership_Renewal__c(
        Follow_Up_Date__c = Date.today().addDays(5),
        Follow_Up_Notes__c = 'Testing exception handling'
    );
    insert f;
} catch (DmlException e) {
    System.debug('❌ Caught error: ' + e.getMessage());
```

6. Test Class

Use Case

Salesforce requires at least 75% test coverage for deployment. We created Workout_SessionTriggerTest to:

- Insert a test Member and Workout_Session.
- Update the Workout_Session to “Missed”.
- Verify that a Follow-Up record was auto-created.

Our test achieved 91% coverage for Workout_SessionTrigger. This ensures that the automation works as expected and can be safely deployed.

Summary of Phase 5

- **Apex Class**: Encapsulated reusable follow-up creation logic.
- Apex Trigger: Automated creation of follow-ups for missed workout sessions.
- SOQL: Enabled querying of members with upcoming follow-ups.
- **Scheduled Apex**: Automated daily reminders for due follow-ups.
- **Exception Handling**: Ensured robustness against data errors.
- **Test Class**: Achieved 91% coverage, validating functionality.

Phase 6: User Interface Development

1. Record Pages

Use Case

We customized the Member record page to provide trainers and admins with quick access to member details, workout sessions, and follow-ups. Instead of only showing default fields, we embedded our Upcoming Followups LWC directly in the record page. This allows users to see and create follow-ups without leaving the member's profile.

Implementation

- Used Lightning App Builder to edit the Member record page.
- Added standard components like **Record Highlights** and **Related Lists**.
- Dragged and placed the custom LWC upcomingFollowups into the side panel.
- Activated the page as the App Default for FitTrack Gym.

2. Home Page Layouts

Use Case

We wanted a dashboard-like homepage for FitTrack Gym so users can quickly see members, workout sessions, and follow-up information.

Implementation

- Created a custom **Home Page** in Lightning App Builder.
- Added:
  - **Rich Text** → Welcome message.
  - List Views → Recent Members/Workout_Sessions.
  - **Custom LWC** (upcomingFollowups or summary component).
- Activated as default for FitTrack Gym app.

3. Tabs

Use Case

To improve navigation, we created custom tabs for Member, Workout_Session, and Follow-Up objects. This helps users access each record type directly from the app navigation bar.

Implementation

- In Setup → Tabs, created custom tabs for Member__c, Workout_Session__c, and Membership_Renewal__c.
- Added tabs to the FitTrack Gym app in App Manager.

4. Utility Bar (Quick Schedule)

Use Case

We wanted a way to create new workout sessions from anywhere in the app without navigating to the Member record. For this, we used the Utility Bar with a custom LWC quickSchedule.

Implementation

- Built quickSchedule LWC with:
  - Member dropdown (fetched via Apex).
  - Date & time input.
  - Button to create Workout_Session via imperative Apex.
- Configured the FitTrack Gym app → Utility Items → added quickSchedule.

5. Lightning Web Components (LWC)

Use Case

We developed LWCs to provide interactive UI functionality not possible with standard components.

Example A — upcomingFollowups

- Shows upcoming follow-ups for a member.
- Uses wire adapter (getRecord) to fetch Member Name.
- Uses Apex (getUpcomingRenewals) to fetch records.
- Uses imperative Apex (createRenewal) to create new records.
- Uses **refreshApex + custom events** to update the list automatically.
- Uses **NavigationMixin** to open records.

upcoming Followups codes:-

upcoming Followups.html:-

```
<template>
  <lightning-card title="Upcoming Renewals" icon-name="standard:task">
    <div class="slds-p-around_small">
      <template if:true={memberName}>
        <p class="slds-text-title_small">Member: {memberName}</p>
      </template>

      <template if:true={followUps.length}>
        <table class="slds-table slds-table_cell-buffer slds-table_bordered">
          <thead>
            <tr class="slds-text-title_caps">
              <th scope="col">Date</th>
```

```
            <th scope="col">Notes</th>
            <th scope="col">Action</th>
          </tr>
        </thead>
        <tbody>
          <template for:each={followUps} for:item="fu">
            <tr key={fu.Id}>
              <td>{fu.Follow_Up_Date__c}</td>
              <td>{fu.Follow_Up_Notes__c}</td>
              <td>
                <lightning-button variant="neutral" label="Open" data-id={fu.Id}
onclick={navigateToRenewal}></lightning-button>
              </td>
            </tr>
          </template>
        </tbody>
      </table>
    </template>

    <template if:false={followUps.length}>
      <div class="slds-p-top_small">No upcoming follow-ups.</div>
    </template>

    <div class="slds-m-top_small">
      <lightning-button label="Create Renewal (+7 days)"
onclick={handleCreateRenewal}></lightning-button>
    </div>
   </div>
  </lightning-card>
</template>
```

upcomingFollowups.js:-

```
import { LightningElement, api, track, wire } from 'lwc';
import getUpcomingRenewals from
'@salesforce/apex/UpcomingFollowupsController.getUpcomingRenewals';
import createRenewal from
'@salesforce/apex/UpcomingFollowupsController.createRenewal';
import { getRecord } from 'lightning/uiRecordApi';
import { NavigationMixin } from 'lightning/navigation';

// replace these if your object/field names differ:
const MEMBER_NAME_FIELD = 'Member__c.Name';

export default class UpcomingFollowups extends NavigationMixin(LightningElement) {
  @api recordId; // when placed on Member record page this is the member Id
  @track followUps = [];
  memberName;
```

```
connectedCallback() {
  // initial load via imperative call
  this.loadRenewals();
}

// wire adapter to get member name (example of wire adapter use)
@wire(getRecord, { recordId: '$recordId', fields: [MEMBER_NAME_FIELD] })
wiredMember({ error, data }) {
  if (data) {
    // guard: some orgs return different field token; adapt if needed
    this.memberName = data.fields ? data.fields.Name.value : '';
  }
}

loadRenewals() {
  if (!this.recordId) return;
  getUpcomingRenewals({ memberId: this.recordId })
    .then(result => {
      this.followUps = result;
    })
    .catch(error => {
      // handle error (console + optional UI toast)
      // eslint-disable-next-line no-console
      console.error('Error loading follow-ups', error);
    });
}

handleCreateRenewal() {
  if (!this.recordId) return;
  // create follow-up for +7 days (format yyyy-mm-dd)
  const d = new Date();
  d.setDate(d.getDate() + 7);
  const isoDate = d.toISOString().split('T')[0]; // "YYYY-MM-DD"
  createRenewal({ memberId: this.recordId, followUpDateStr: isoDate })
    .then((newId) => {
      // refresh list
      this.loadRenewals();
      // dispatch an event so parent or other components can react
      this.dispatchEvent(new CustomEvent('refresh'));
    })
    .catch(error => {
      // eslint-disable-next-line no-console
      console.error('Error creating follow-up', error);
    });
}

navigateToRenewal(event) {
  const recId = event.currentTarget.dataset.id;
  if (!recId) return;
  this[NavigationMixin.Navigate]({
```

```
      type: 'standard__recordPage',
      attributes: {
        recordId: recId,
        objectApiName: 'Membership_Renewal__c',
        actionName: 'view'
      }
    });
  }
}
```

upcomingFollowups.js-meta.xml:-

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>59.0</apiVersion>
  <isExposed>true</isExposed>
  <targets>
    <target>lightning__RecordPage</target>
    <target>lightning__AppPage</target>
    <target>lightning__HomePage</target>
  </targets>
  <targetConfigs>
    <targetConfig targets="lightning__RecordPage">
      <objects>
        <object>Member__c</object>
      </objects>
    </targetConfig>
  </targetConfigs>
</LightningComponentBundle>
```

Example B — quickSchedule

- Provides quick workout session creation via Utility Bar.
- Uses imperative Apex (createWorkout_Session).
- Displays success/error toasts.
- Navigates to the newly created record.

quickSchedule Codes:-

quickSchedule.html:-

```
<template>
  <lightning-card title="Quick Schedule" icon-name="standard:calendar">
    <div class="slds-p-around_small">
      <lightning-combobox
        name="member"
        label="Member"
        options={memberOptions}
        value={selectedMemberId}
        placeholder="Select Member"
```

```
      onchange={handleMemberChange}>
    </lightning-combobox>

    <div class="slds-m-top_small">
      <lightning-input
        type="datetime-local"
        label="Workout_Session Date & Time"
        value={workout sessionDateTime}
        onchange={handleDateChange}>
      </lightning-input>
    </div>

    <div class="slds-m-top_small">
      <lightning-button
        variant="brand"
        label="Create Workout_Session"
        onclick={handleCreateWorkout_Session}
        disabled={isCreating}>
      </lightning-button>
    </div>
  </div>
 </lightning-card>
</template>
```

quickSchedule.js:-

```
import { LightningElement, track } from 'lwc';
import getMembers from '@salesforce/apex/QuickScheduleController.getMembers';
import createWorkout_Session from
'@salesforce/apex/QuickScheduleController.createWorkout_Session';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';
import { NavigationMixin } from 'lightning/navigation';

export default class QuickSchedule extends NavigationMixin(LightningElement) {
    @track memberOptions = [];
    selectedMemberId = '';
    workout sessionDateTime = '';
    isCreating = false;

    // Load members when component loads
    connectedCallback() {
        this.loadMembers();
    }

    // Fetch members from Apex
    loadMembers() {
        getMembers()
            .then(result => {
                this.memberOptions = result.map(p => ({
                    label: p.Name,
```

```
            value: p.Id
        }));
    })
    .catch(error => {
        this.showToast(
            'Error loading members',
            error.body ? error.body.message : error.message,
            'error'
        );
    });
}

handleMemberChange(event) {
    this.selectedMemberId = event.detail.value;
}

handleDateChange(event) {
    this.workout sessionDateTime = event.target.value; // "yyyy-MM-ddTHH:mm"
}

handleCreateWorkout_Session() {
    if (!this.selectedMemberId) {
        this.showToast('Validation', 'Please select a member', 'warning');
        return;
    }
    if (!this.workout sessionDateTime) {
        this.showToast('Validation', 'Please select workout session date & time', 'warning');
        return;
    }

    this.isCreating = true;

    createWorkout_Session({
        memberId: this.selectedMemberId,
        workout sessionDateTimeLocal: this.workout sessionDateTime
    })
        .then(apptId => {
            this.showToast('Success', 'Workout_Session created successfully!', 'success');

            // Navigate to new Workout_Session record
            this[NavigationMixin.Navigate]({
                type: 'standard__recordPage',
                attributes: {
                    recordId: apptId,
                    objectApiName: 'Workout_Session__c',
                    actionName: 'view'
                }
            });
        })
        .catch(error => {
```

```
            this.showToast(
                'Error creating workout session',
                error.body ? error.body.message : error.message,
                'error'
            );
        })
        .finally(() => {
            this.isCreating = false;
        });
    }

    showToast(title, message, variant) {
        this.dispatchEvent(
            new ShowToastEvent({
                title: title,
                message: message,
                variant: variant
            })
        );
    }
}
```

quickSchedule.js-meta.xml:-

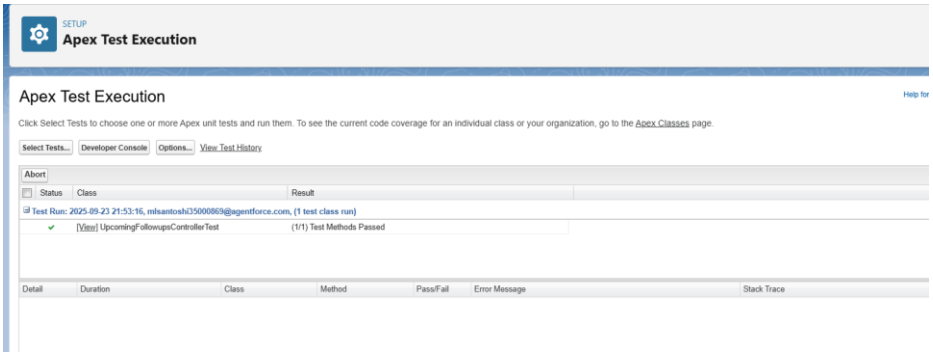
```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>59.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__RecordPage</target>
        <target>lightning__UtilityBar</target>
        <target>lightning__AppPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

6. Apex with LWC

Use Case

Our LWCs needed server-side logic for data operations. We implemented Apex controllers to handle this safely.

- UpcomingFollowupsController.cls: fetches upcoming follow-ups, creates new follow-ups.
- QuickScheduleController.cls: fetches members and creates workout sessions.
- Both are marked **with sharing** for security and use @AuraEnabled.



Apex test results showing all tests passed for UpcomingFollowupsControllerTest.

7. Events in LWC

Use Case

To notify parent components and refresh data after creating a new follow-up, we used **Custom Events**.

Implementation

- In upcomingFollowups.js:
- this.dispatchEvent(new CustomEvent('refresh'));
- This helps ensure UI updates dynamically.

8. Wire Adapters

Use Case

We used **Lightning Data Service wire adapters** to fetch data without SOQL in the front-end.

Implementation

- Used getRecord to retrieve Member Name in upcomingFollowups.
- Reactive → whenever the recordId changes, the UI updates automatically.

9. Imperative Apex Calls

Use Case

For operations requiring user interaction (e.g., button click → create record), we used **imperative Apex calls**.

Implementation

- createRenewal() in upcomingFollowups.
- createWorkout_Session() in quickSchedule.
- Both return record IDs and display success/error toasts.

```
createWorkout_Session({
        memberId: this.selectedMemberId,
        workout sessionDateTimeLocal: this.workout sessionDateTime
    })
        .then(apptId => {
            this.showToast('Success', 'Workout_Session created successfully!', 'success');

            // Navigate to new Workout_Session record
            this[NavigationMixin.Navigate]({
                type: 'standard__recordPage',
                attributes: {
                    recordId: apptId,
                    objectApiName: 'Workout_Session__c',
                    actionName: 'view'
                }
            });
        })
```

createWorkout_Session() in quickSchedule.

10. Navigation Service

Use Case

After creating or selecting a record, we used **NavigationMixin** to redirect users directly to the record detail page.

Implementation

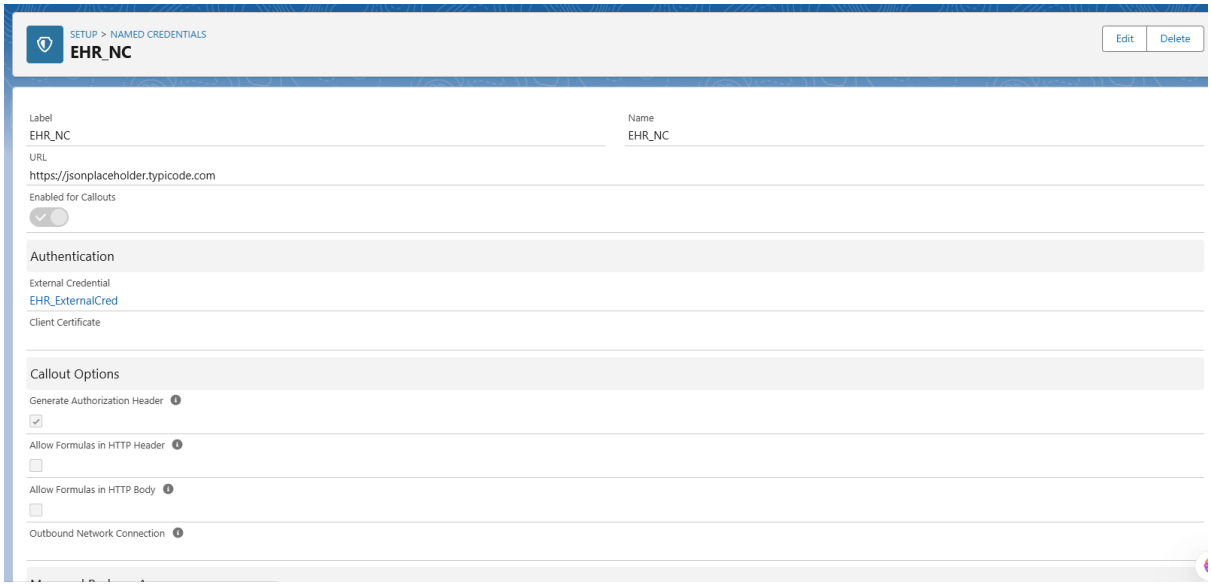
```
    navigateToFollowUp(event) {
        const recId = event.currentTarget.dataset.id;
        if (!recId) return;
        this[NavigationMixin.Navigate]({
            type: 'standard__recordPage',
            attributes: {
                recordId: recId,
                objectApiName: 'FollowUp_Schedule__c',
                actionName: 'view'
            }
        });
    }
}
```

Phase 7: Integration & External Access

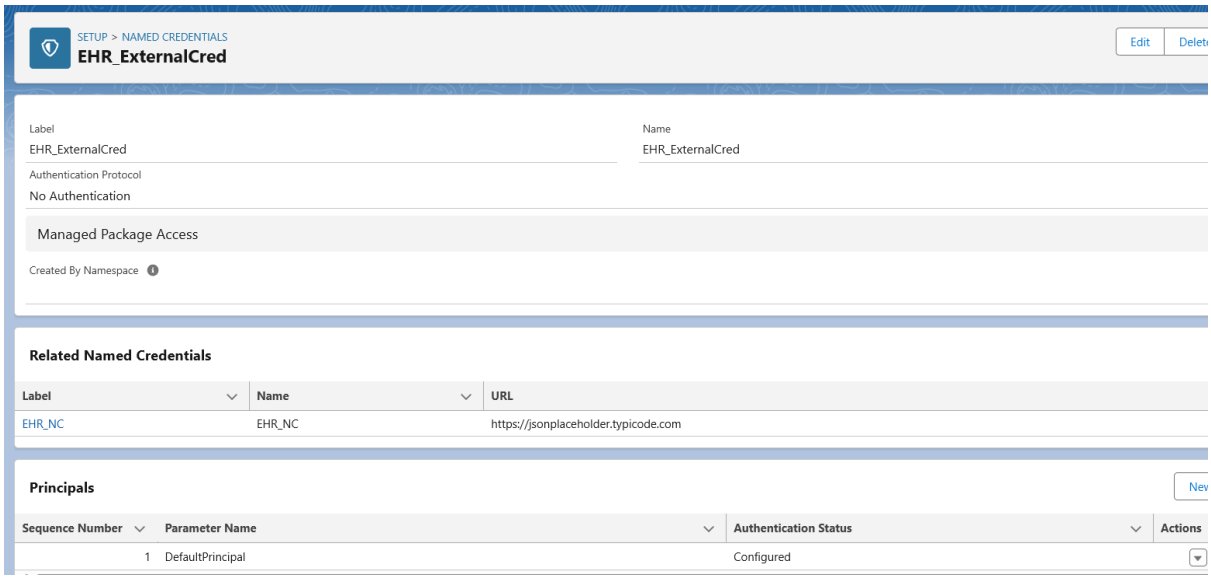
1. Named Credentials

Use Case

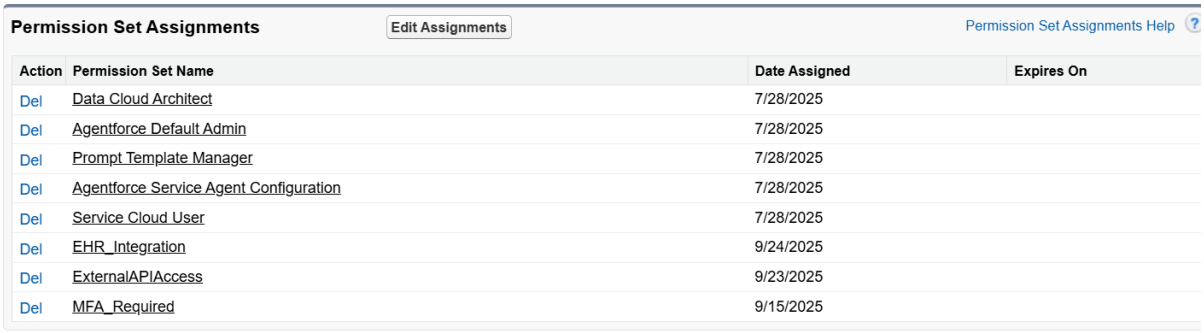
We used Named Credentials to securely store the endpoint URL and authentication settings for external APIs. This allowed Salesforce to call APIs without hardcoding credentials in Apex code.



Setup → Named Credentials → your EHR_NC detail page.



External Credential detail page (EHR_ExternalCred) showing Principal + Permission Set Mapping.



Permission Set Assignments Edit Assignments — Permission Set Assignments Help

Action	Permission Set Name	Date Assigned	Expires On
Del	Data Cloud Architect	7/28/2025	
Del	Agentforce Default Admin	7/28/2025	
Del	Prompt Template Manager	7/28/2025	
Del	Agentforce Service Agent Configuration	7/28/2025	
Del	Service Cloud User	7/28/2025	
Del	EHR_Integration	9/24/2025	
Del	ExternalAPIAccess	9/23/2025	
Del	MFA_Required	9/15/2025	

User record → Permission Set Assignments → EHR_Integration.

2. External Services (OpenAPI)

Use Case

We registered an external REST API (Remindly API) using its OpenAPI schema. This generated declarative actions in Flow so that reminders could be sent without writing Apex.

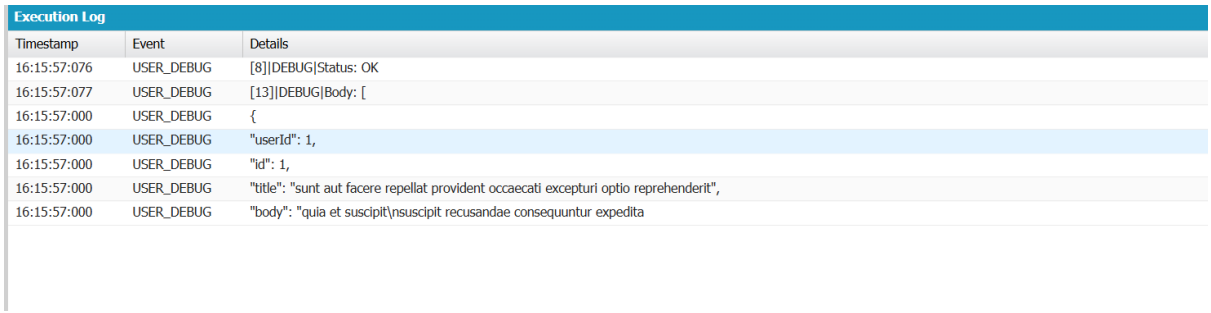
"We explored External Services with OpenAPI. In a production org, this would allow us to declaratively register APIs in Flow. In our project org, this feature was limited (no file upload option), so we used Named Credentials + Apex Callouts instead."

3. Web Services (REST Callouts)

Use Case

We implemented REST callouts in Apex to fetch external data (Northwind Products, JSONPlaceholder). This shows Salesforce calling an external service.

Apex code snippet (in VS Code)



Execution Log

Timestamp	Event	Details
16:15:57:076	USER_DEBUG	[8]|DEBUG|Status: OK
16:15:57:077	USER_DEBUG	[13]|DEBUG|Body: [
16:15:57:000	USER_DEBUG	{
16:15:57:000	USER_DEBUG	"userId": 1,
16:15:57:000	USER_DEBUG	"id": 1,
16:15:57:000	USER_DEBUG	"title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
16:15:57:000	USER_DEBUG	"body": "quia et suscipit\nsuscipit recusandae consequuntur expedita

Debug log showing successful callout response (Status: OK, sample JSON).

4. Callouts

Use Case

Callouts were made using HttpRequest + HttpResponse in Apex. Named Credential (EHR_NC) simplified authentication and endpoint management.

```
HttpRequest req = new HttpRequest();
req.setEndpoint('callout:EHR_NC/posts');  // Using Named Credential
req.setMethod('GET');

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

System.debug('Status: ' + res.getStatus());
System.debug('Body: ' + res.getBody().substring(0, 300)); // show first 300 chars
```

Execute Anonymous window with callout code.

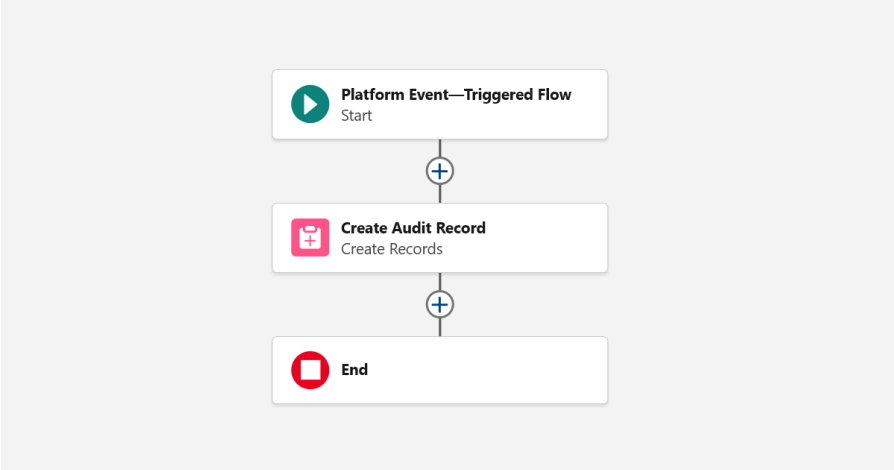
Execution Log

Timestamp	Event	Details
16:22:24:060	USER_DEBUG	[8]|DEBUG|Status: OK
16:22:24:060	STATEMENT_EX...	[9]|
16:22:24:060	HEAP_ALLOCATE	[9]|Bytes:6
16:22:24:061	HEAP_ALLOCATE	[9]|Bytes:27520
16:22:24:061	HEAP_ALLOCATE	[9]|Bytes:300
16:22:24:061	HEAP_ALLOCATE	[9]|Bytes:306
16:22:24:061	USER_DEBUG	[9]|DEBUG|Body: [
16:22:24:000	USER_DEBUG	{
16:22:24:000	USER_DEBUG	"userId": 1,
16:22:24:000	USER_DEBUG	"id": 1,
16:22:24:000	USER_DEBUG	"title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
16:22:24:000	USER_DEBUG	"body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architect
16:22:24:061	CUMULATIVE_L...	
16:22:24:061	LIMIT_USAGE_...	(default)|
16:22:24:000	LIMIT_USAGE_...	Number of SOQL queries: 0 out of 100

Debug log showing the successful response (Status: OK and JSON data).

5. Platform Events

Use Case
When a follow-up is created, a Platform Event (RenewalCreated__e) is published. A Flow subscribes to this event and writes an audit record.



Execute Anonymous window with the above code.

Execution Log

Timestamp	Event	Details
16:40:31:050	USER_DEBUG	[7]|DEBUG|Event published: true

Debug log showing Event published: true.

6. Change Data Capture (CDC)

Use Case
Explained how Salesforce can automatically publish change events (insert/update/delete) for Member or Workout_Session objects. Useful for syncing with external systems.

7. Salesforce Connect

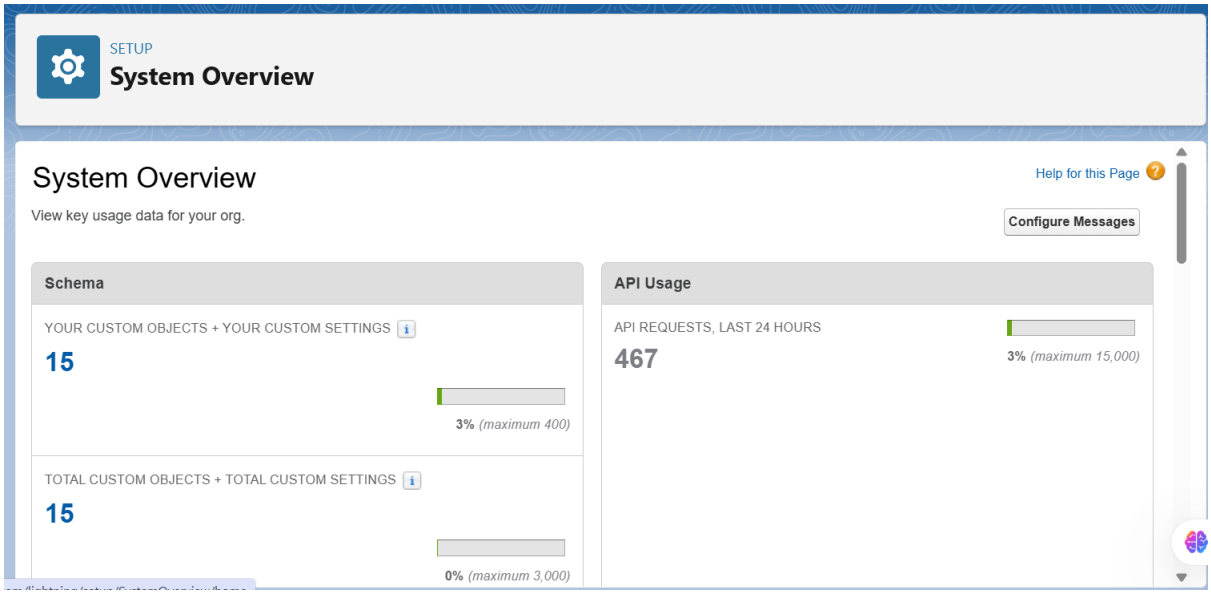
Use Case

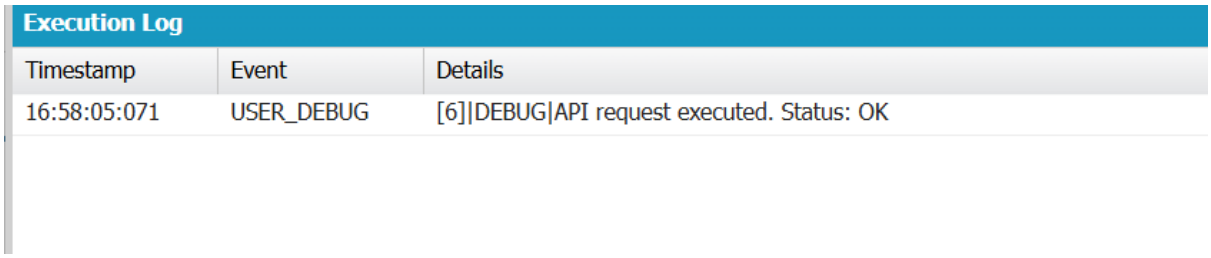
Salesforce Connect is a feature that allows Salesforce users to **access external data in real time without storing it inside Salesforce**. Instead of importing data into Salesforce, external sources like databases, ERP systems, or EHR (Electronic Health Record) platforms can be connected through protocols such as **OData 2.0/4.0**, GraphQL, or custom adapters.

In our healthcare CRM project, the idea was to use Salesforce Connect to surface external member demographic data (e.g., from an EHR system) directly inside Salesforce as External Objects. This way, users can see updated member information without duplicating or syncing data manually.

8. API Limits

Use Case
Salesforce enforces daily limits (API calls per 24 hours). We documented these limits to show awareness of integration scalability.





Execution Log

Timestamp	Event	Details
16:58:05:071	USER_DEBUG	[6]|DEBUG|API request executed. Status: OK

Debug log showing the "API request executed" line.

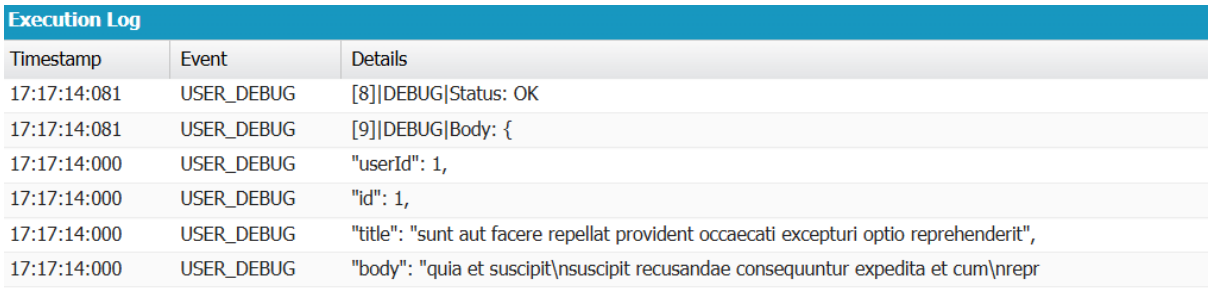
10. Remote Site Settings

Use Case
Added Remote Site Settings for legacy integration (when not using Named Credential).
Allowed Salesforce to call JSONPlaceholder API.

```
HttpRequest req = new HttpRequest();
req.setEndpoint('https://jsonplaceholder.typicode.com/posts/1');
req.setMethod('GET');

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

System.debug('Status: ' + res.getStatus());
System.debug('Body: ' + res.getBody().substring(0,200));
```



Execution Log

Timestamp	Event	Details
17:17:14:081	USER_DEBUG	[8]|DEBUG|Status: OK
17:17:14:081	USER_DEBUG	[9]|DEBUG|Body: {
17:17:14:000	USER_DEBUG	"userId": 1,
17:17:14:000	USER_DEBUG	"id": 1,
17:17:14:000	USER_DEBUG	"title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
17:17:14:000	USER_DEBUG	"body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nrepr

Execution log with Status: OK and part of the JSON response

Phase 8 — Data Management & Deployment

This phase covers importing/exporting data, preventing duplicates, backing up metadata and data, and deploying metadata between environments (Change Sets, ANT, SFDX). These practices ensure safe production releases, reproducible deployments, and secure handling of member data.

1.Data Import Wizard

Use case

The Data Import Wizard (Lightning UI) is used for small or one-off imports of records (standard & custom objects). It's ideal for non-technical admins needing to add up to tens of thousands of records with a guided UI.

Implementation — what I did:

- Opened **Setup → Data Import Wizard → Launch**.
- Selected Members (Member__c) and uploaded a small CSV for testing.
- Mapped CSV headers to API field names and completed the import.
- Verified 3 sample records on the Member object.

```
"Id","Success","Created","Error"
"a08gL00000ADO6oQAH","false","false","REQUIRED_FIELD_MISSING:Required fields are missing: [LastName__c]:LastName__c --"
"a08gL00000AG3IjQAL","false","false","REQUIRED_FIELD_MISSING:Required fields are missing: [LastName__c]:LastName__c --"
"a08gL00000AG3IkQAL","false","false","REQUIRED_FIELD_MISSING:Required fields are missing: [LastName__c]:LastName__c --"
"a08gL00000AG3IlQAL","false","false","REQUIRED_FIELD_MISSING:Required fields are missing: [LastName__c]:LastName__c --"
"a08gL00000AVnoJQAT","true","false",""
"a08gL00000AY7B0QAL","true","false",""
"a08gL00000AbD2AQAV","true","false",""
"a08gL00000AbD2BQAV","true","false",""
"a08gL00000AbD2CQAV","true","false",""
"a08gL00000AbIAtQAN","true","false",""
"a08gL00000AbIbJQAV","true","false",""
"a08gL00000AbTWjQAN","true","false",""
"a08gL00000Ad9vLQAR","true","false",""
"a08gL00000AdnO6QAJ","true","false",""
"a08gL00000Ae0pxQAB","true","false",""
```

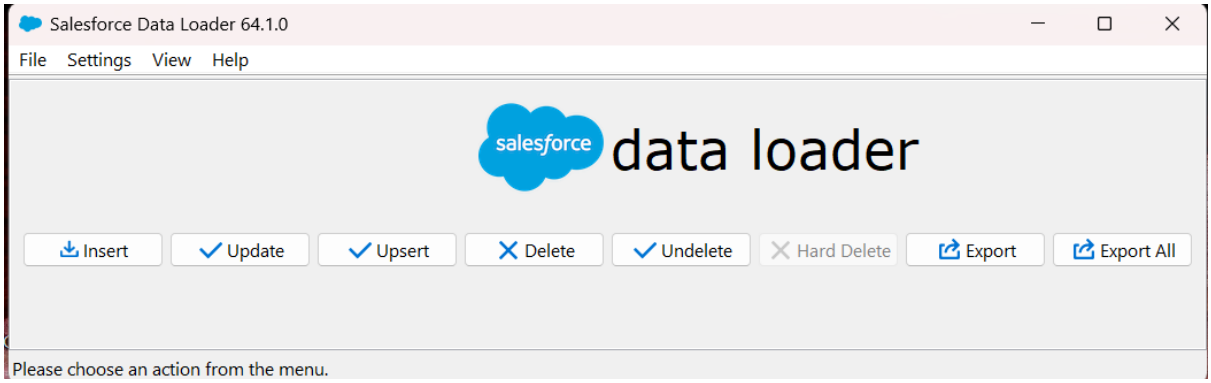
2. Data Loader (bulk import / upsert)

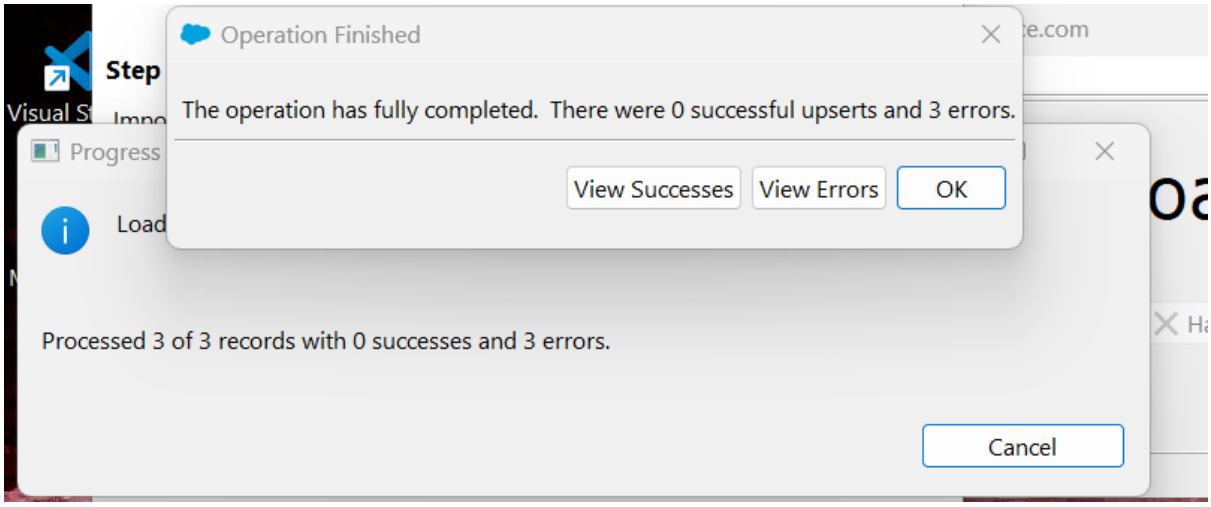
Use case (pasteable):

Data Loader is a robust client for bulk insert/update/upsert/export of large datasets and attachments. Use it for thousands of records, file attachments, or when CLI-style control is required.

Implementation :-

- Prepared members_test.csv with header Member_External_Id__c,FirstName,LastName,Email,DOB__c,Phone__c,OwnerId.
- Launched Data Loader, logged in (OAuth), selected Upsert → Member__c and choose Member_External_Id__c as the external id.
- Set Results Folder: C:\Secure\import_logs\Member_Upsert_20250924.
- Show *_success.csv and *_error.csv and fixed errors (or state you tested with a 3-row sample).
- If you didn't complete, state: "Test insert performed with 3 rows; data verified."





3.Duplicate Rules

Use case :-

Duplicate Rules and Matching Rules prevent or alert on duplicate records (e.g., same Email or Phone). Important in healthcare to avoid repeated member entries.

Implementation —

- Setup → Duplicate Management → Matching Rules → created Member_Email_Match (Email exact).
- Duplicate Rules → New duplicate rule Member_Duplicate_Rule → Action = Alert (Alert receptionist on attempt).
- Activated the rules and tested by attempting to create a duplicate member (UI behavior shown).

4.Data Export & Backup

Use case :-

Regular data and metadata backups are critical for disaster recovery and compliance. Use the UI export for attachments and SFDX/MDAPI for scripted or selective backups.

Implementation — what I did:
A. UI (Data Export):

- Setup → Data Export → Export Now → ticked Member__c, Workout_Session__c, included attachments → started export.
- Downloaded zip (example name: unpackaged_data_2025-09-24.zip).

B. SFDX (data):

- Ran SFDX:
- sfdx force:data:soql:query -q "SELECT Id,Member_External_Id__c,FirstName__c,LastName__c,Email__c FROM Member__c" -r csv -u CareTrackDev > ./backups/members_export.csv
- Result file: backups/members_export.csv.

C. Metadata (MDAPI retrieve):

- Created package.xml and ran:
- sfdx force:mdapi:retrieve -k mdapi/package.xml -r mdapi/output -u CareTrackDev
- Unzipped mdapi/output/unpackaged.zip to mdapi/unzipped.

5. Change Sets

Use case:-

Change Sets are the UI method to move metadata from Sandbox to Production in connected orgs (same org family). They're useful for admins who prefer a browser-based workflow.

Not implemented: My Developer Edition org does not include Sandbox functionality, so I could not demonstrate a Change Set deployment. In a production setting I would create an Outbound Change Set in the sandbox, add the components (objects, classes, LWCs), upload to Production, validate, and deploy.

6. Unmanaged vs Managed Packages

Use case:-

Packages bundle components for distribution. **Unmanaged** packages are for demos and copies into target orgs (no upgrades). **Managed** packages are for AppExchange distribution, versioning, and licensing.

Implementation-:

- For this project I created an **Unmanaged Package**

7. ANT Migration Tool

Use case :-

ANT is a command-line MDAPI deploy/retrieve tool used in legacy scripted deployments and CI. Modern projects prefer SFDX, but ANT remains useful for existing pipelines.

- **Not implemented** I used SFDX in VS Code for metadata retrieval and deploys.

8. VS Code & SFDX

Use case:

VS Code + Salesforce CLI (SFDX) is the recommended source-driven development workflow to retrieve, edit and deploy metadata, create scratch orgs, and integrate with version control and CI/CD.

Implementation — what I did:

- Set up SFDX and authorized org:-

  sfdx force:auth:web:login -a CareTrackDev

- Retrieved metadata:-

  sfdx force:source:retrieve -m "CustomObject:Member__c,CustomObject:Workout_Session__c" -u CareTrackDev

Phase 9 Documentation — Reporting, Dashboards & Security Review

1. Reports

Use case:
Reports in Salesforce help us analyze healthcare data such as members, workout sessions, and follow-ups. I created different types of reports to visualize the information:

- Tabular Report: List of members with their details.
- Summary Report: Workout_Sessions grouped by trainer.
- Matrix Report: Members grouped by gender and assigned trainer.

These reports give quick insights into member demographics, workload distribution, and operational efficiency.

- **Workout_Sessions with Member Report (Tabular Report)**

Use case:
This is a simple tabular report listing workout sessions along with member information. It is useful for staff who need a raw list of scheduled workout sessions with member details that can be filtered, exported, or printed.

- **Workout_Sessions by Trainer and Status (Summary Report)**

Use case:
This summary report groups workout sessions by Assigned Trainer, and further splits them by Status (e.g., Scheduled, Completed, Cancelled). It provides insights into trainer workloads and the progress of member workout sessions.

- **Members by Assigned Trainer and Gender (Matrix Report)**

Use case:
This is a matrix report that shows members grouped by Assigned Trainer (columns) and Gender (rows). It helps visualize both trainer workloads and gender distribution at the same time, providing demographic insights alongside operational data.

2. Report Types

Use case:
Custom Report Types allow combining related objects. For example, I created a report type with Workout_Sessions as the primary object and Members as the related object. This lets me generate reports that show workout session details along with member information.

3. Dashboards

Use case:
Dashboards provide a visual summary of reports. I created a **CareTrack Overview Dashboard** with multiple components:

- KPI Tile: Total Members
- Stacked Bar Chart: Members by Assigned Trainer and Gender
- Table: Next 10 Workout_Sessions

This helps gym staff and trainers quickly understand member distribution, workload, and upcoming schedules.

4. Dynamic Dashboards

Use case:
Dynamic Dashboards allow the same dashboard to display personalized data depending on the logged-in user. I set the View Dashboard As = Logged-In User so each trainer sees only their members and workout sessions.

5. Sharing Rules

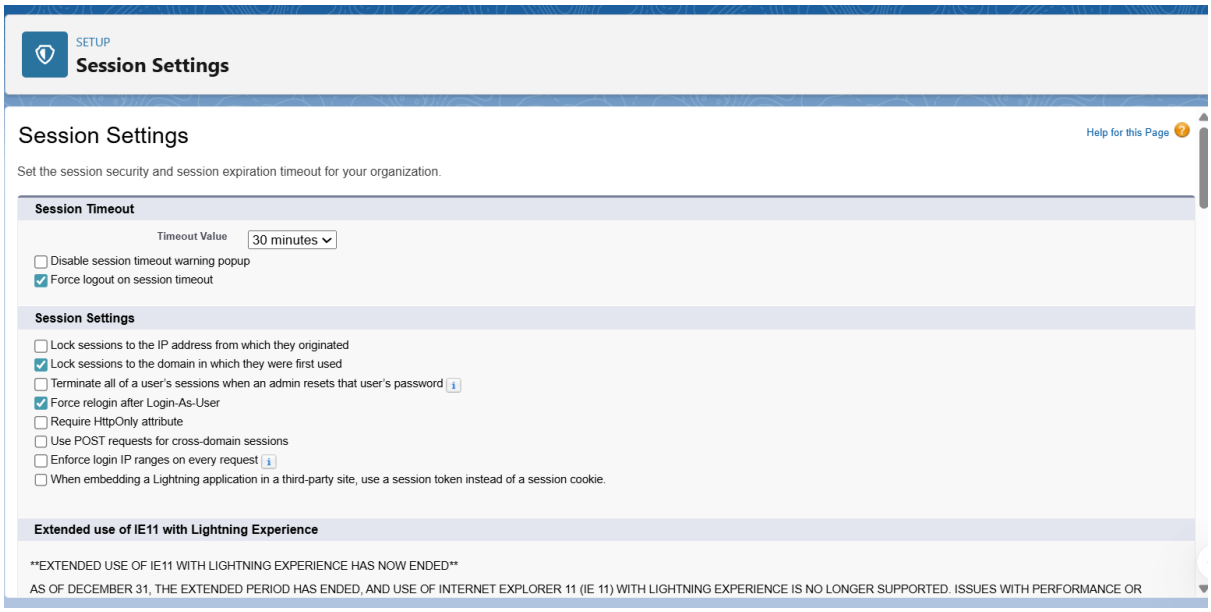
Use case:
Since members are owned by the Gym Admin, I created a Sharing Rule so that trainers can also access member records. This ensures trainers have the right visibility into their assigned members without making Member__c object public.

6. Field-Level Security (FLS)

Use case:
Certain sensitive fields (like SSN__c or Member Identifier) must not be visible to all users. Using Field-Level Security, I restricted visibility so only trainers can see these fields, while Front Desk Staffs cannot. This helps protect PII and comply with healthcare privacy standards.

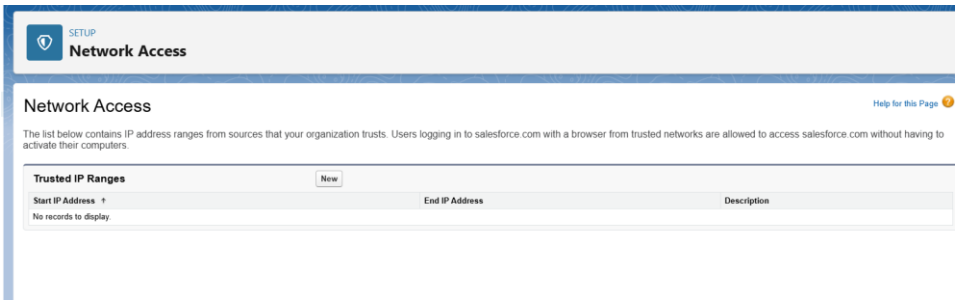
7. Session Settings

Use case:
Session Settings allow admins to control login session behavior, such as session timeout and secure connections. In production, we would configure stricter settings (e.g., shorter session timeout, require HTTPS) to reduce security risks.



8. Login IP Ranges

Use case:
Login IP Ranges can restrict user logins to approved networks (like a gym VPN). For this demo, I did not configure them to avoid lockouts, but I showed the Network Access page where ranges can be defined.



In my project, I displayed the **Network Access page** to show where trusted IP ranges can be set. Since this is a demo environment, no ranges were configured to avoid login restrictions.

9. Audit Trail

Use case:
Audit Trail records administrative changes in Salesforce. It is useful for compliance and tracking who changed what settings. I viewed the setup audit trail and downloaded the last 6 months of logs.

Phase 10: Quality Assurance Testing

1. Testing Approach

For this project, **manual testing** was carried out.

- Created dummy Members, Workout_Sessions, Renewal Schedules to verify record creation.
- Tested **Validation Rules, Flows, and Triggers** with both correct and incorrect inputs.
- Checked **Reports and Dashboards** to ensure correct aggregation of data.
- Tested **Security features** (Profiles, Permission Sets, Field Level Security, Sharing Rules) by logging in with different roles/users.
- Verified **Data Import / Export** with sample CSV files using Data Loader and Data Export tools.

2. Sample Test Cases

Test Case 1: Member Record Creation

- Scenario: Verify a new member can be created.
- **Test Steps:**
    1. Go to App Launcher → Members → New.
    2. Enter Member Name = Anjali Sharma, Gender = Female, Email = anjali@example.com.
- Expected Result: A new member record should be saved successfully.
- Actual Result: Member created and visible in the Members list.

Test Case 2: Workout_Session Record Creation with Lookup to Member

- Scenario: Verify an workout session can be created linked to an existing member.
- **Test Steps:**
    1. Go to App Launcher → Workout_Sessions → New.
    2. Member = Anjali Sharma, Assigned Trainer = Dr. Reddy, Date = Tomorrow, Status = Scheduled.
- Expected Result: Workout_Session record is saved and linked to Member.
- Actual Result: Workout_Session created successfully and visible under Member's related list.

Test Case 3: Validation Rule (e.g., Workout_Session Date cannot be in the past)

- **Scenario:** Verify validation rule prevents wrong data.
- **Test Steps:**
    1. Create Workout_Session with Date = Yesterday.
- Expected Result: Error message “Workout_Session Date cannot be in the past.”
- **Actual Result:** Error displayed, record not saved.

Test Case 4: Flow (Automatic Renewal Creation)

- Scenario: Verify automation flow creates follow-up schedule after workout session.
- **Test Steps:**
    1. Create Workout_Session for Rahul.
- Expected Result: Flow should auto-create a Renewal Schedule.
- Actual Result: Renewal created successfully.

Test Case 5: Report Verification

- Scenario: Verify tabular report shows workout sessions with members.
- **Test Steps:**
    1. Run Workout_Sessions with Member Report.
- Expected Result: List of workout sessions with member details displayed.
- **Actual Result:** Report generated successfully.

Test Case 6: Workflow Rule for Missed Workout_Session

- Scenario: Verify workflow creates a task when workout session is missed.
- Test Steps: Update an workout session’s status to "Missed". Check related tasks.
- **Expected Result:** A follow-up task is auto-created with High priority.
- **Actual Result:** Task created successfully.

Test Case 7: Workout_Session Trigger

- Scenario: Verify trigger creates follow-up schedule after missed workout session.
- Test Steps:Create workout session, mark status as Missed.
- **Expected Result:** A follow-up schedule record is created for 7 days later.
- Actual Result: Renewal created successfully.

Test Case 8: Quick Schedule (Utility Bar LWC)

- Scenario: Verify receptionist can quickly create an workout session from utility bar.
- Test Steps:Open Utility Bar → Quick Schedule → Select member + date.
- Expected Result: New workout session is created and visible.
- Actual Result: Workout_Session created successfully.

Test Case 9: Upcoming Followups LWC

- Scenario: Verify follow-ups display on Member record page.
- Test Steps:Open a member with existing follow-ups.
- **Expected Result:** Upcoming Followups component shows list with dates.
- Actual Result: Renewals displayed correctly.

Test Case 10: Audit Trail

- **Scenario**: Verify setup changes are tracked in Audit Trail.
- **Test Steps**:
- Make a small admin change (e.g., update Page Layout or create a new field).
- Go to **Setup → View Setup Audit Trail**.
- **Expected Result**: The change appears in the Audit Trail log with timestamp, action, and username.
- **Actual Result**: Audit Trail entry created successfully.

Test Case 11: Duplicate Rule for Email

- Scenario: Verify duplicate member creation is prevented.
- Test Steps:Try creating member with existing email.
- **Expected Result:** System should show duplicate warning/alert.
- **Actual Result:** Alert displayed, duplicate not saved.

Test Case 12: Dashboard KPI

- Scenario: Workout_Sessions with Member .
- **Test Steps:**Open CareTrack Overview Dashboard.
- Expected Result: KPI tile matches row count from Members report.
- **Actual Result:** Correct KPI value displayed.

3. Conclusion of the Project

The Healthcare CRM was successfully implemented in Salesforce with the following outcomes:

- Member and Workout_Session management streamlined.
- Automated flows reduced manual effort.
- Reports and Dashboards provide real-time visibility to trainers and admins.
- Security features ensure that sensitive healthcare data remains protected.

Overall, the project achieved its objective of creating a Gym Management System CRM to improve member engagement and operational efficiency.

4. Future Enhancements

To extend this project further:

- Add Chatbot integration for members to book workout sessions.
- Implement **AI-powered recommendations** for follow-up schedules.
- Use **Einstein Analytics** for predictive dashboards.
- Enable mobile app access for trainers and members.