Smart Student Enrollment & Progress Tracker

Problem Statement

Currently, the student enrollment process is completely manual. Students have to physically submit forms or email their requests. They cannot track the status of their enrollment, leading to confusion and delays. Faculty members must manually review and approve requests, which is timeconsuming. There is also no proper reporting mechanism for administrators to monitor student progress or enrollment statistics.

Phase 1 — Problem Understanding

The first phase focuses on clearly understanding the problem we are solving. We defined the issue, wrote user stories, and drew a simple process flow.

- Problem Statement: Enrollment is manual, students can't track progress, and there is no reporting for Admin.

- User Stories: 1) As a Student, I want to request enrollment in a course so I can participate. 2) As a Faculty member, I want to approve/reject enrollment requests so that only eligible students are enrolled. 3) As an Admin, I want to see reports of student enrollment and progress to monitor performance.

- Process Flow Diagram: Student → Enrollment Request → Faculty Approval → Enrollment Status Updated → Admin Reporting.

- **Requirement Gathering:**
  We began by understanding the current enrollment process at universities. The main issues identified were that enrollment was fully manual, students had no way to track the status of their requests, and faculty had to handle approvals through emails or offline communication. This created confusion, delays, and duplication of work. To resolve this, we gathered requirements from the perspective of all users. Finally, we decided Salesforce automation could simplify and centralize the process.

- **Stakeholder Analysis:**
  The project involves three main stakeholders. Students want an easy way to request enrollment in courses and view their progress. Faculty need a structured method to approve or reject requests without managing everything manually. Admins require access

to reports and dashboards to monitor student performance and enrollment trends. By analyzing these stakeholders, we defined clear roles and responsibilities. This ensured that our solution would be user-focused.

- **Business Process Mapping:**
  We mapped out the entire enrollment process to visualize how information flows. The process starts when a Student submits an enrollment request. It moves to Faculty for approval or rejection. Once approved, the enrollment status is updated to Enrolled. Finally, Admins access this data in reports and dashboards. This clear mapping helped us identify the key objects required — Student, Course, and Enrollment. We used a flow diagram to illustrate this.

- **Industry-specific Use Case Analysis:**
  We studied how real-world universities and institutions handle enrollments. Most rely on either manual paperwork or fragmented systems that do not give students transparency. Our analysis confirmed the need for a digital solution to improve efficiency, reduce errors, and provide visibility. With Salesforce's CRM and automation features, we could design a solution similar to what many modern universities are adopting. This validated our approach.

- **AppExchange Exploration:**
  Before starting custom development, we explored Salesforce AppExchange for readymade solutions. While there are education management apps available, they were either too complex or not aligned with our specific use case. Many had extra features we didn't need, which would add unnecessary complexity. After this exploration, we decided to build a custom solution from scratch. This ensured simplicity and full control over functionality.

Verification:

We confirmed that we could explain the problem in two clear sentences and had a process diagram ready. This ensures the team fully understands the goal before starting development.

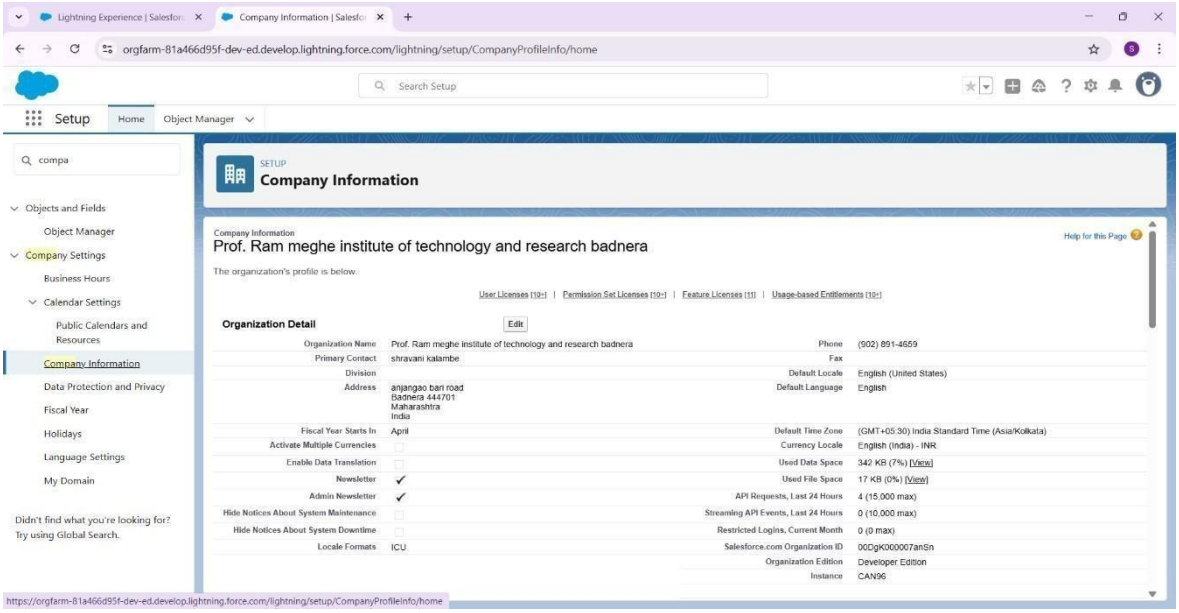
Phase 2 — Org Setup & Security

The second phase involves preparing the Salesforce org to support our solution. This includes setting up organization details, business hours, holidays, roles, profiles, and security settings.

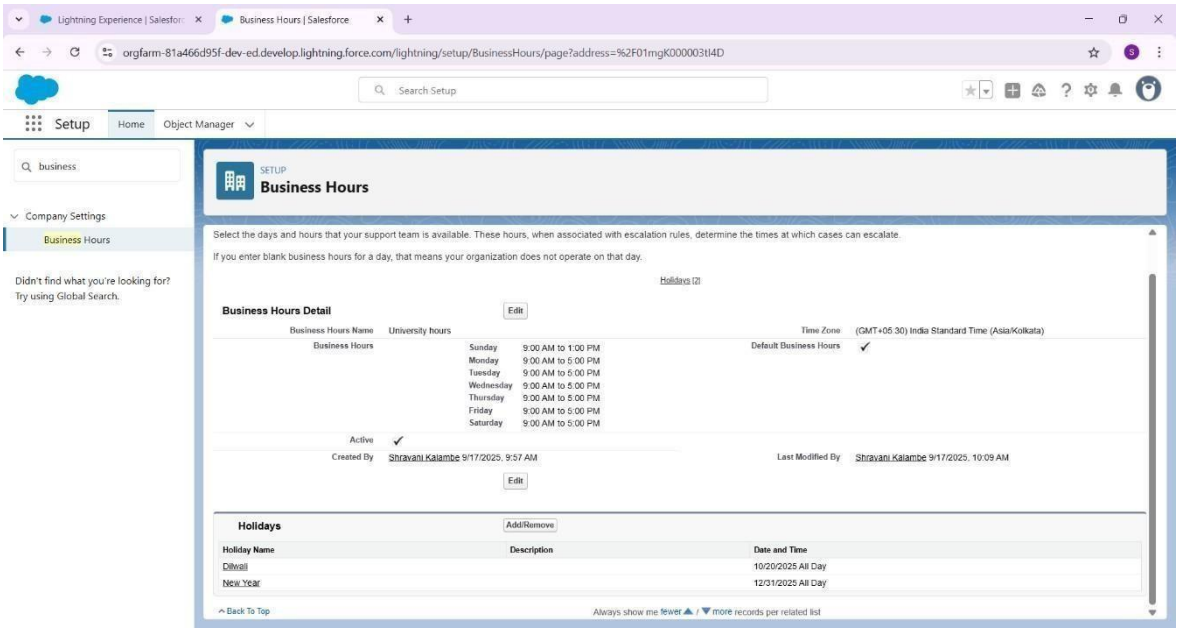
Salesforce Editions:

Salesforce provides different editions like Essentials, Professional, Enterprise, and Developer. For this project, we selected the **Developer Edition** because it offers all customization features at no cost. This edition allows us to create custom objects, write Apex triggers, build Lightning Components, and configure automation flows.

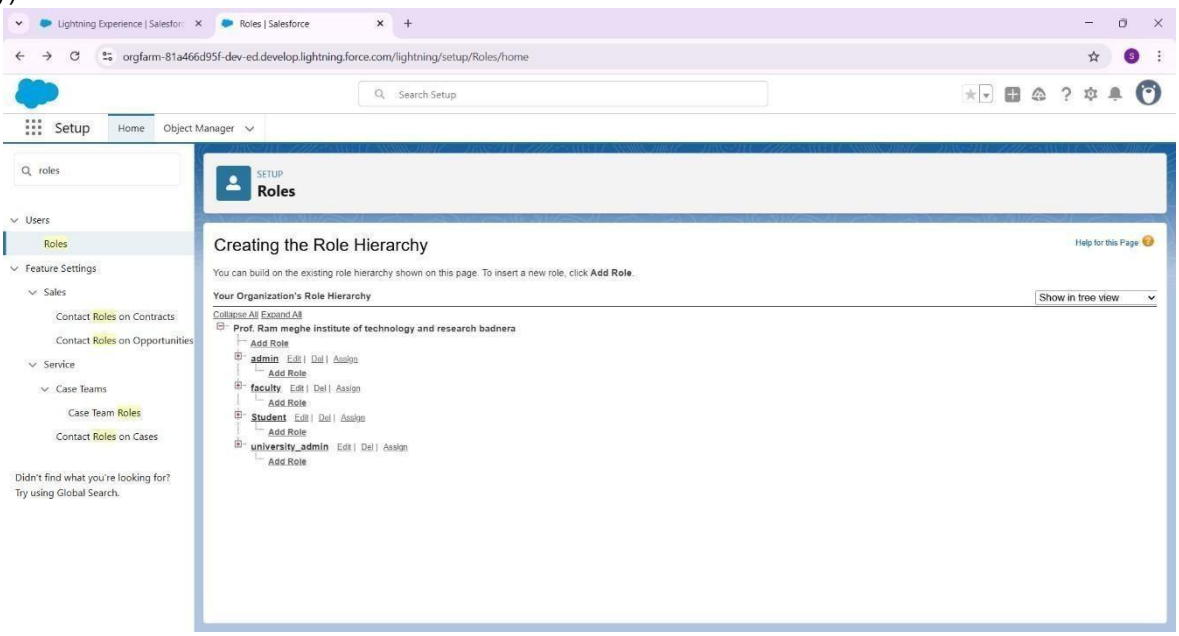
- **Company Profile Setup** : Configured Company Information: Set organization name to ABC University, timezone to Asia/Kolkata, and currency to INR.



Business Hours & Holidays: To ensure system automations respect working times, we configured Business Hours. We created “University Hours” as Monday to Friday, 9 AM–5 PM. Next, we added two holidays — Diwali and Christmas. These holidays were associated with University Hours so that approval processes and escalation rules would pause during non-working days. This ensures the system operates realistically according to the academic calendar.



Roles: We created sample users to represent real stakeholders. Prof. Rahul was added as a Faculty member with a Salesforce license, which grants full CRM access. Student One was created with a Salesforce Platform license, which provides limited access but is ideal for selfservice students. Defining correct licenses ensures users only consume the necessary features. This step also allowed us to simulate real scenarios by logging in as different roles.Defined Roles: Admin (top level), Faculty (reports to Admin), and Student (reports to Faculty).

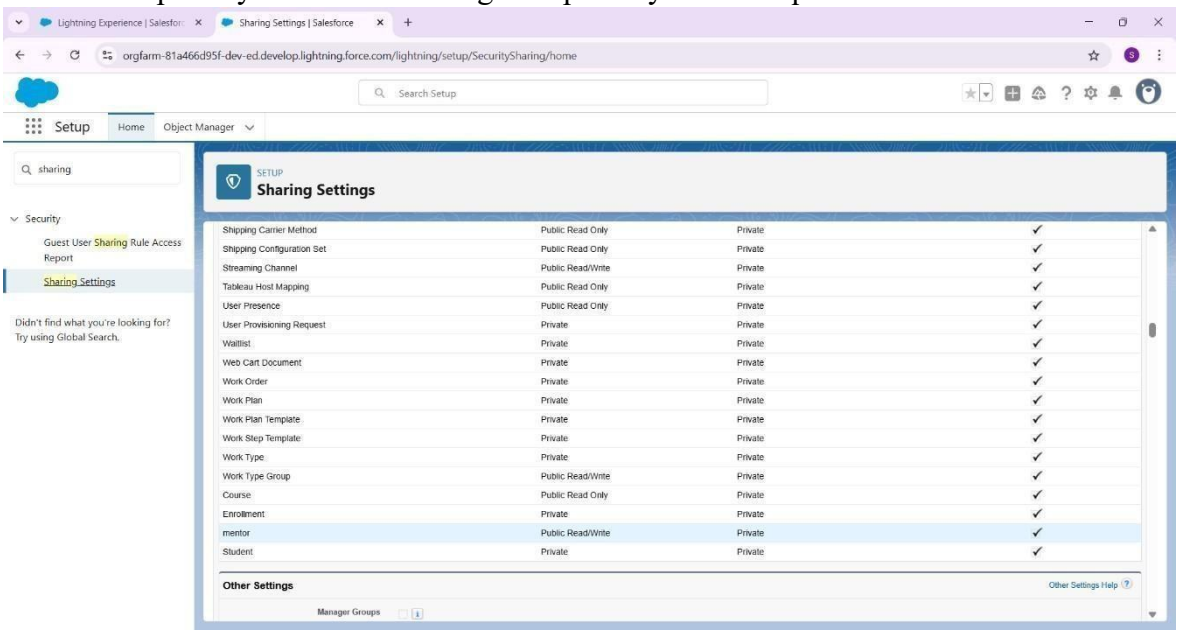


Profiles: Created Profiles: Cloned Standard User to create Faculty and Student profiles with appropriate permissions.

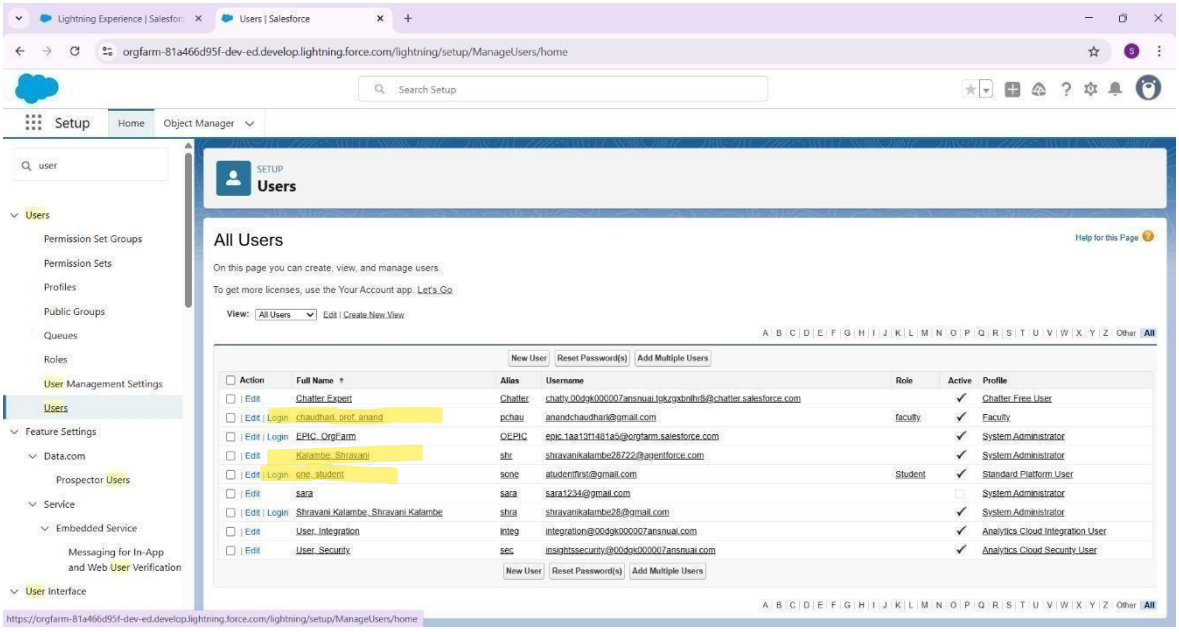
- Configured Sharing Settings: Student__c = Private **OWD (Org-Wide Defaults):** Org-Wide Defaults establish the baseline visibility for all objects. We configured:

- Student__c = Private (a student can only see their own record).
- Course__c = Public Read Only (everyone can view courses).
- Enrollment__c = Controlled by Parent (visibility depends on the linked Student and Course).

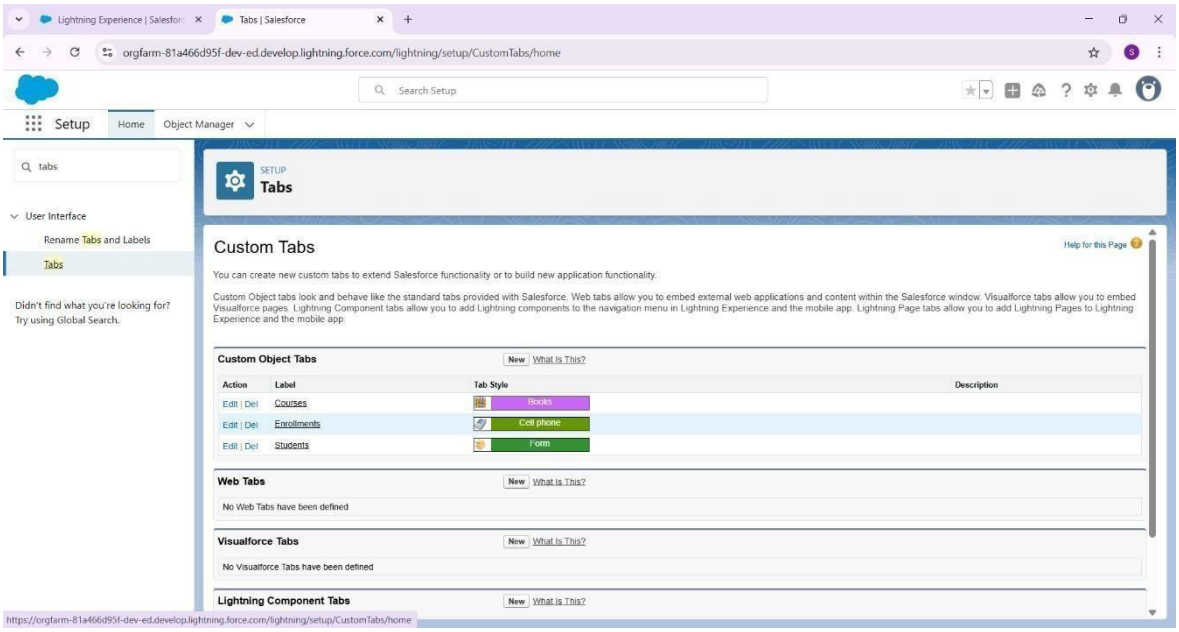
This ensures data privacy while maintaining transparency where required.



User Setup & Licenses: Created Users: Prof. Rahul (Faculty profile) and Student One (Student profile) with respective roles and licenses.



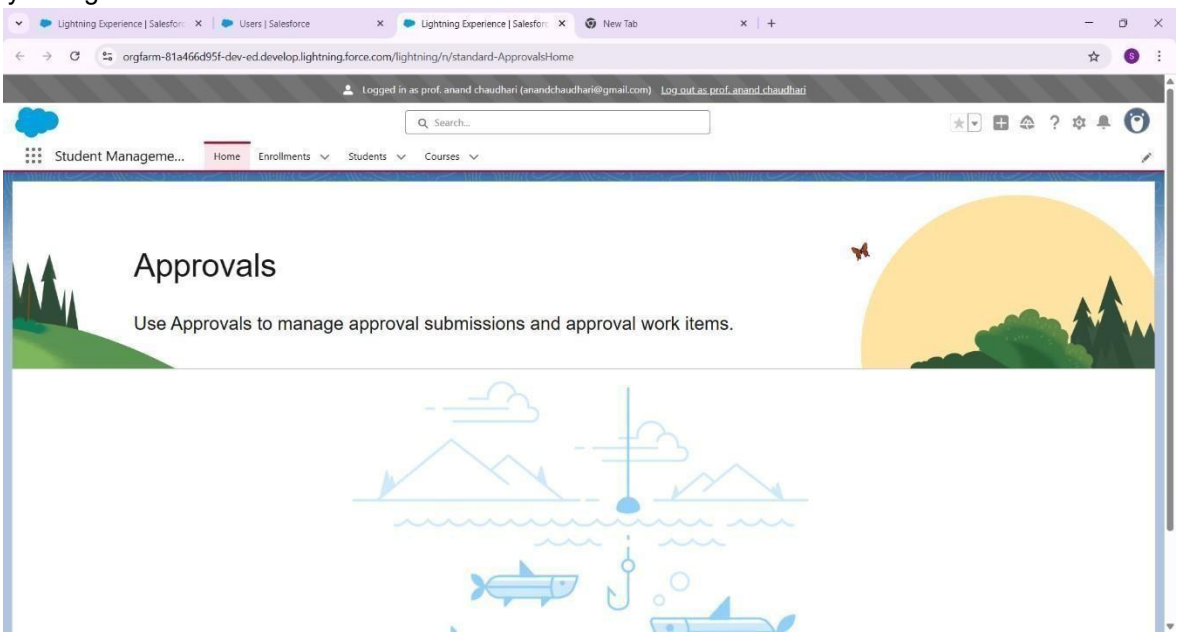
- Created Custom Tabs: Student, Course, and Enrollment objects now have tabs to allow navigation in the Lightning app.



- Added Tabs to Student Management App: Ensured Students, Courses, and Enrollments are visible to Admin and Faculty.

Login Access Policies:

Salesforce allows admins to log in as other users to test profiles and permissions. We used this feature to verify the experience of both Faculty and Students. Logging in as Prof. Rahul confirmed he could view Courses and Enrollments but not other Students. Logging in as Student One confirmed they only had access to their own data. This final step ensured our security configuration was correct..



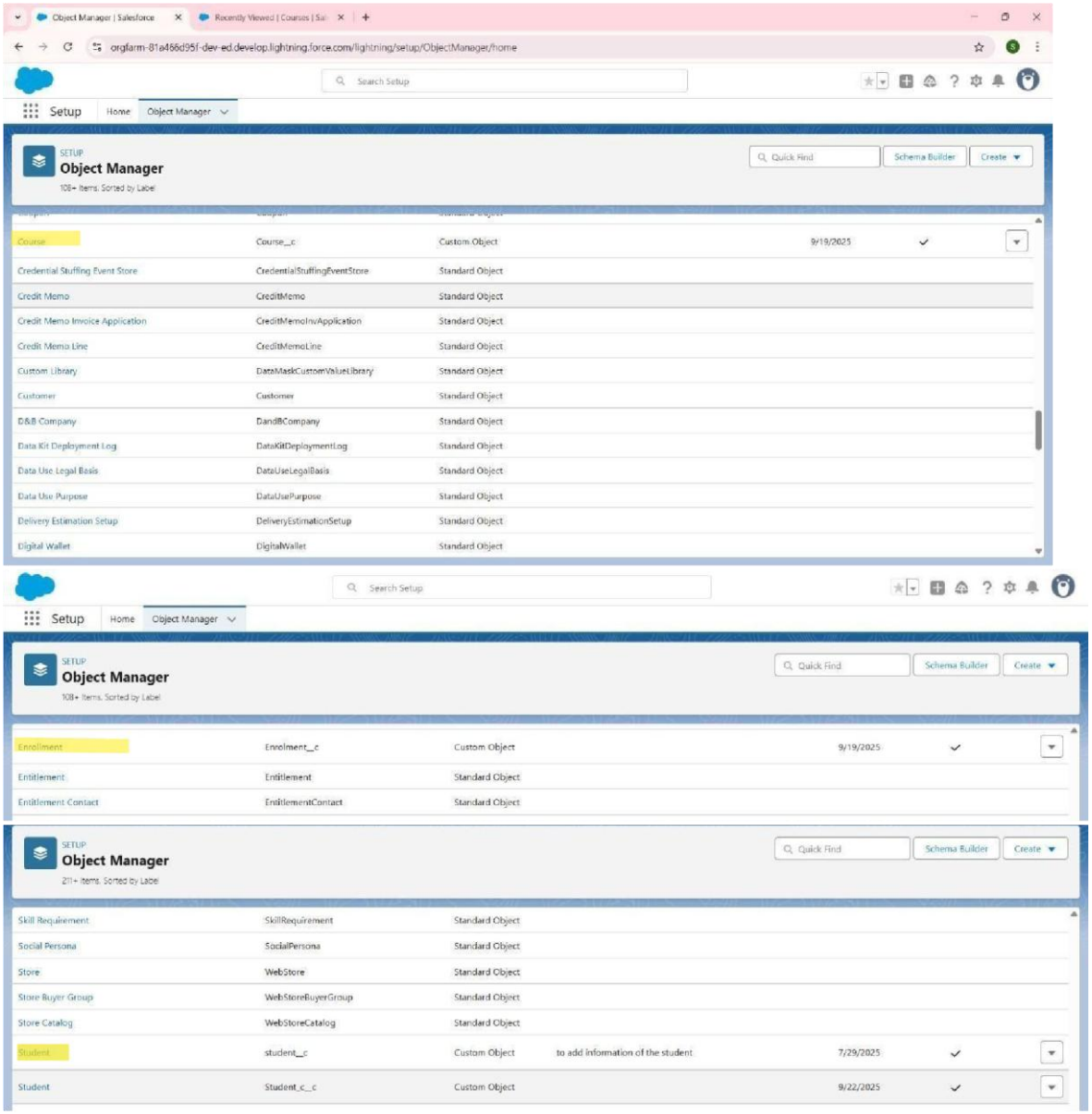
Verification:

Logged in as Faculty and confirmed they could see the Courses and Enrollments tabs but not other students' records. Logged in as Student and confirmed they could only see their own data. This ensured security settings worked as expected.

Phase 3 — Data Modeling & Relationships

1. Standard & Custom Objects

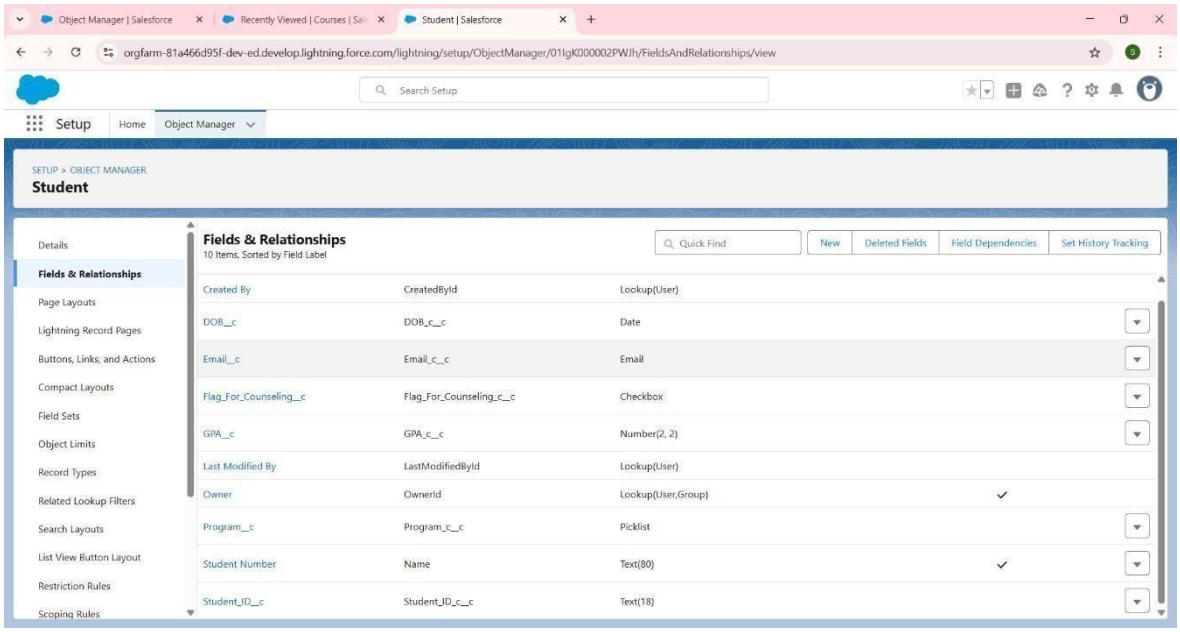
Information: Objects are database tables.
What We Did: Created 3 custom objects: Student__c, Course__c, Enrollment__c. **Verification:** Objects appear in Object Manager.



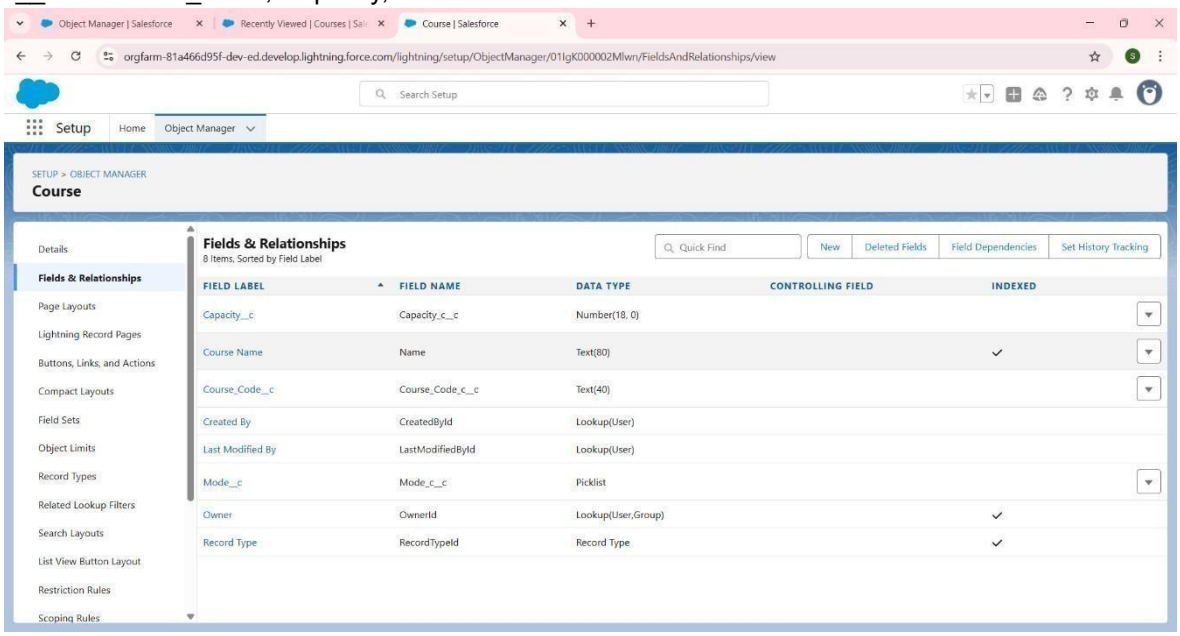
2. Fields

Information: Store attributes of each object.

- Student__c → Student_ID, DOB, Program, GPA, Email



- Course__c → Course_Code, Capacity, Mode.

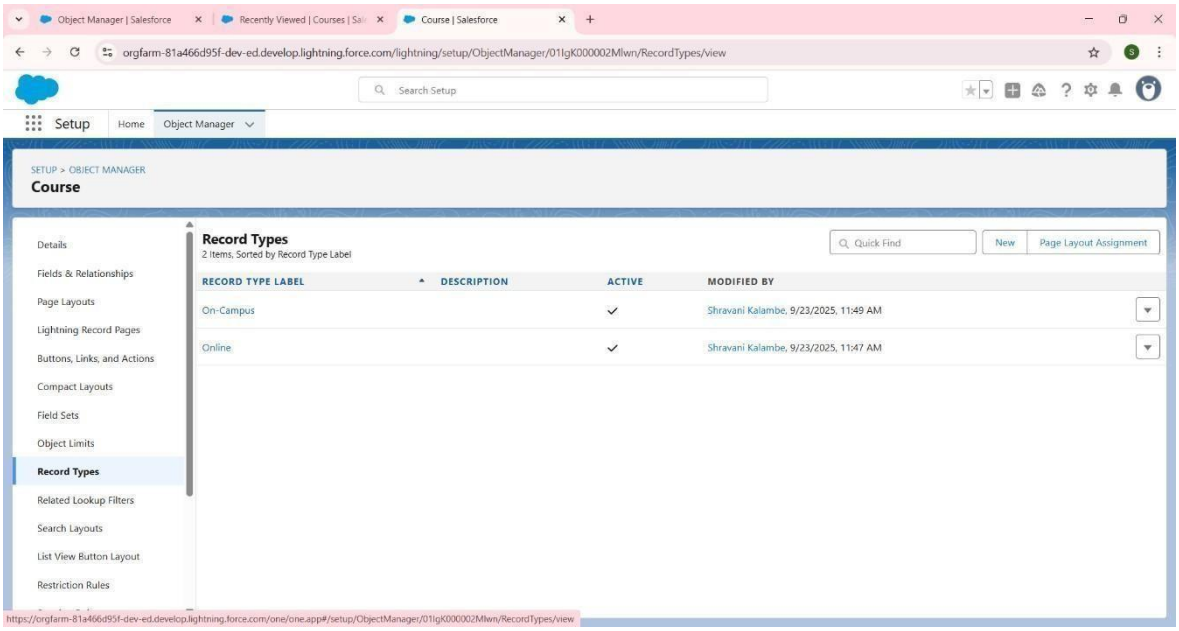


- Enrollment__c → Status, Request Date, Student__c (lookup), Course__c (lookup).
  Verification: Fields visible on page layouts and records.

3. Record Types

Information: Different business processes per object.

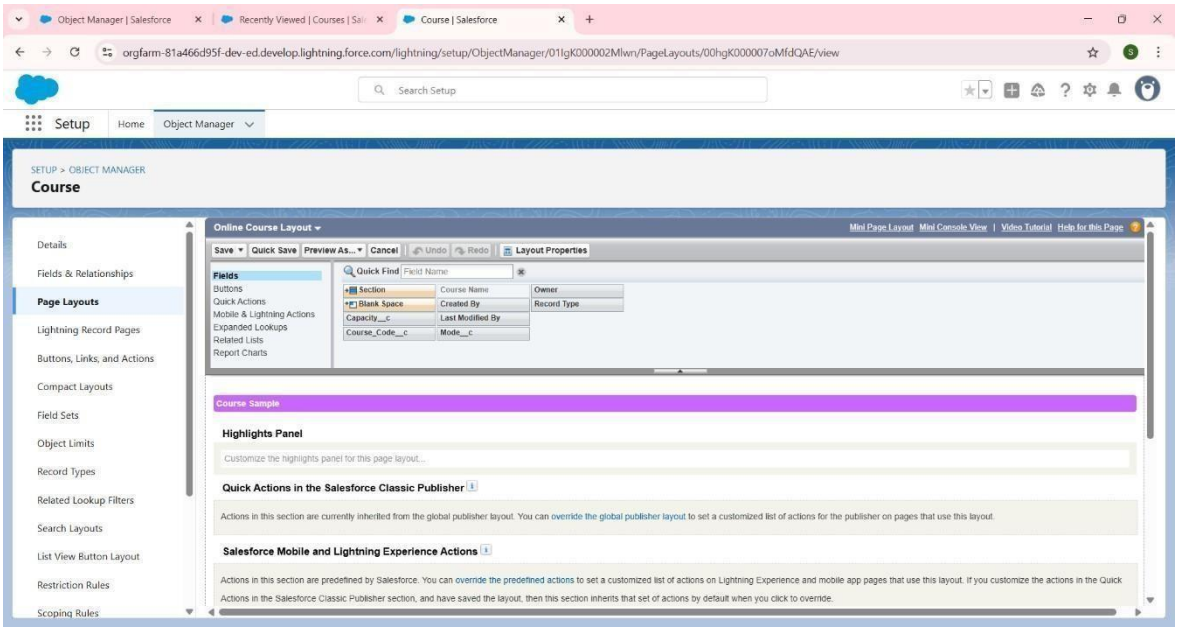
For Course__c → Record Types: Online, On-Campus.
Verification: When creating Course, Salesforce asks for record type → correct layout shown.



4. Page Layouts

Information: Define UI for records.

- Online Course Layout → fields like Platform, Meeting Link.
- On-Campus Layout → fields like Classroom, Campus Location. **Verification:** Different layouts show based on record type.



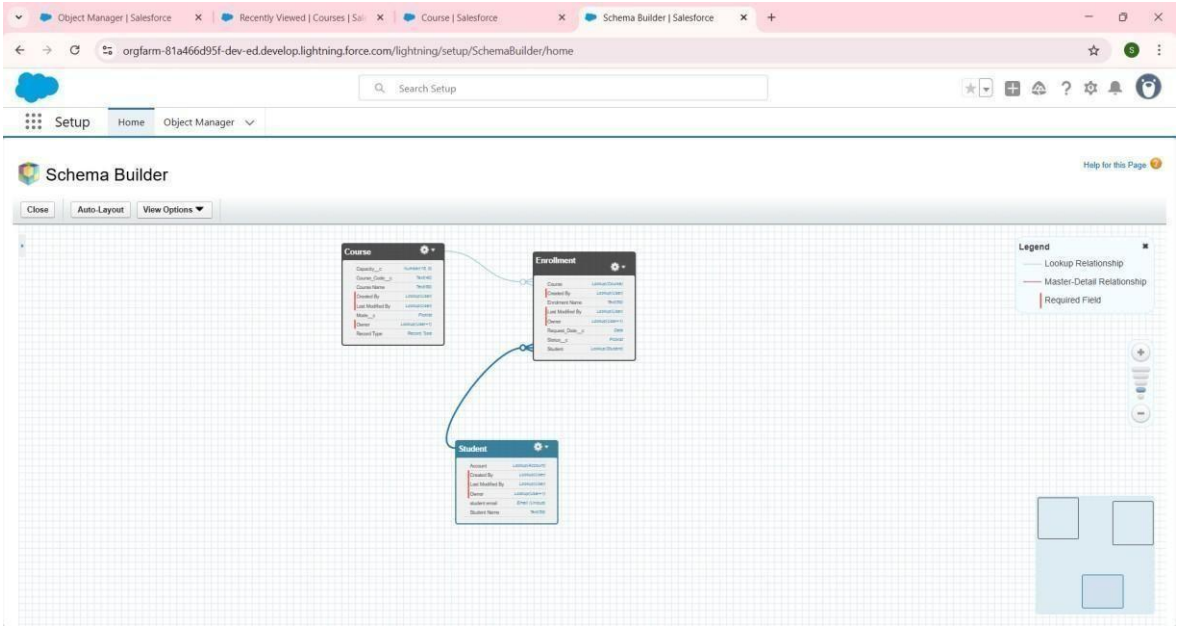
5. Compact Layouts

Information: Show key fields in highlights panel.

Student Compact Layout = Name, Program, GPA.
Verification: Compact layout visible on top of record page.

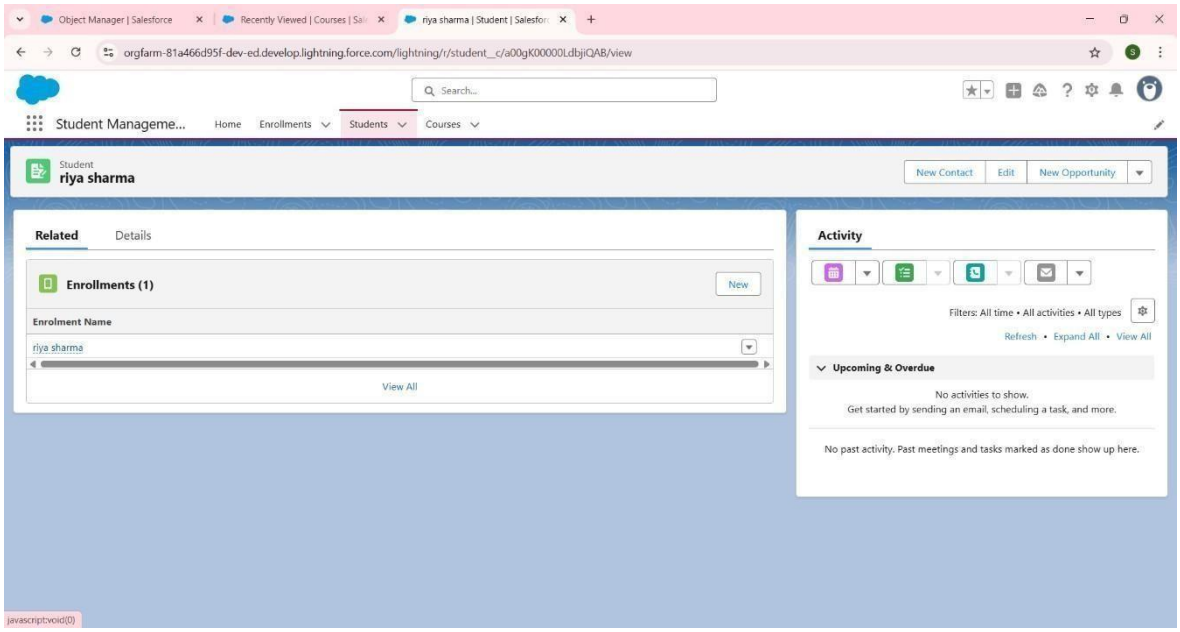
6. Schema Builder

Information: Visual tool to see object relationships.
What We Did: Used Schema Builder to check links between Student, Course, Enrolment.
Verification: Diagram shows Enrollment linked to Student & Course.



7. Lookup vs Master-Detail vs Hierarchical

Information: Relationship types.
Used **Lookup** relationships for Enrollment → Student and Enrollment → Course. **Verification:** Enrollment records display in related lists on Student & Course.



8. Junction Objects

Information: Object to connect 2 objects in many-to-many. Enrollment__c acts as junction between Student & Course.
Verification: One Student can enroll in many Courses; One Course can have many Students.

9. External Objects

Information: Used for connecting external DBs. Not used in this project. **Verification:** N/A.

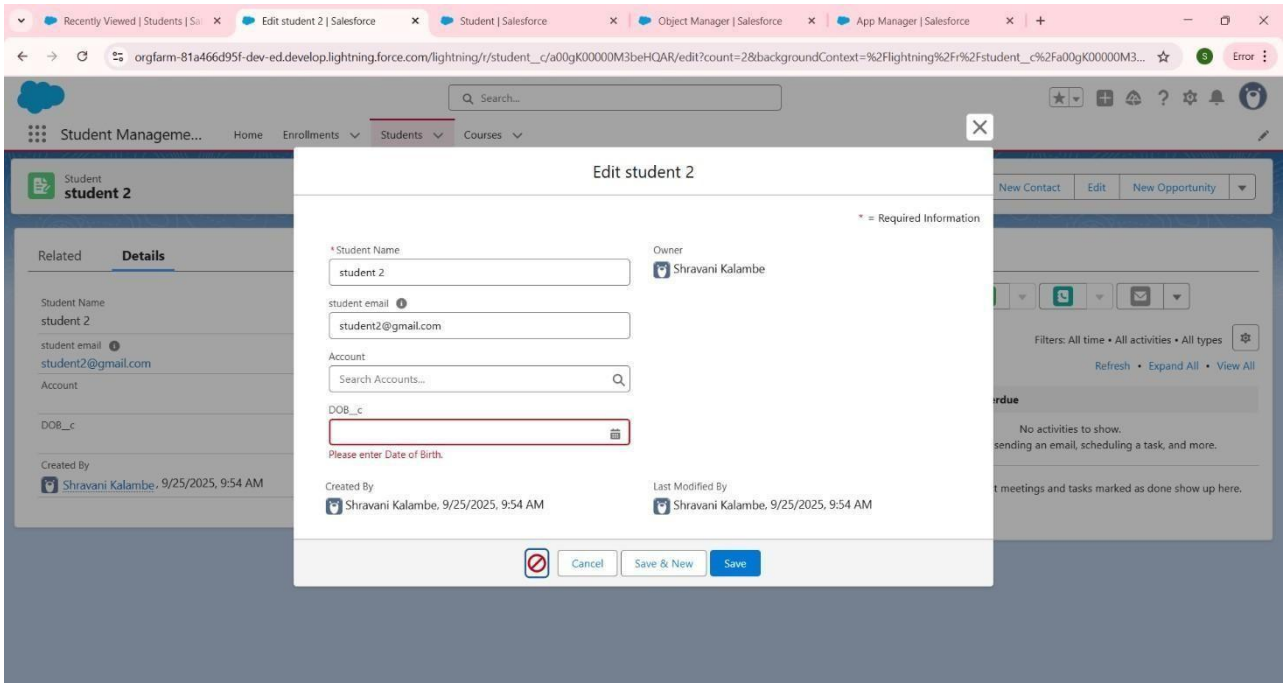
End of Phase 3 Verification:

- Created Student record.
- Created Course record.
- Created Enrollment linking Student + Course. • Related lists working. • Record Types tested.

Phase 4: Process Automation (Admin)

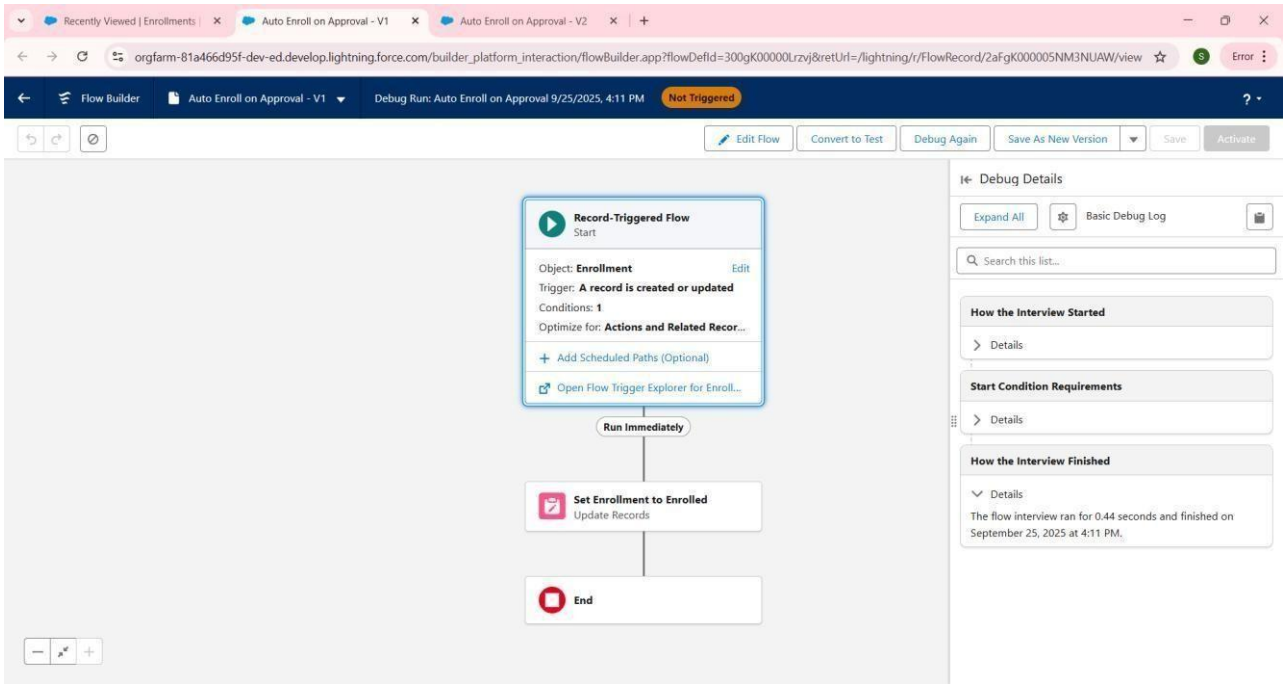
1. Validation Rules

We implemented a **Validation Rule** on the Student__c object to ensure that the **Date of Birth (DOB__c)** is mandatory. This helps maintain accurate student records and prevents saving incomplete data. The rule uses the formula ISBLANK(DOB__c) and shows an error message if left empty. This runs automatically when a record is created or edited.



2. Workflow Rules

Although **Workflow Rules** are legacy tools, we planned to use modern alternatives like **Flows**. Traditionally, workflow rules help automate actions like field updates, email alerts, or tasks based on record criteria. In our project, we replaced workflows with **Record-Triggered Flows** and **Approval Processes** for better flexibility. If needed, simple workflows can still be added for quick alerts or updates.



3. Process Builder

Salesforce is phasing out **Process Builder** in Favor of **Flows**, so we did not use it directly. Instead, all process automation (like auto-enrolment after approval) was implemented via **Flows**. Process Builder is useful for multi-step logic and was previously used for approval actions, field updates, and related record changes. In our solution, its functionality is fully replaced by Flow Builder.

4. Approval Process

An **Approval Process** was created for Enrollment__c to manage enrollment requests. When the status is set to "Requested", it gets routed to a user in the **faculty role** for review. Upon approval, the status automatically updates to "Approved"; if rejected, it changes to "Rejected". This ensures formal authorization before enrolling a student into a course.

5. Flow Builder

We used a **Record-Triggered Flow** on Enrollment__c to detect when a record is approved. It automatically updates the status to **"Enrolled"** if the previous status wasn't already approved. Flow Builder replaced older tools like Workflow and Process Builder. More Flow types (Screen, Auto-launched, Scheduled) can be added later for GPA recalculation or student re-evaluation.

6. Email Alerts

While not implemented directly yet, **Email Alerts** can be used to notify students or faculty. For example, an alert can be sent to a student when their enrollment is approved or rejected. This can be set up within an **Approval Process** or **Flow** using predefined templates and recipients. It enhances communication without manual effort.

7. Field Updates

We used **Field Updates** as part of the Approval Process to change the Status__c field. On **final approval**, the status is updated to "Approved", and on **rejection**, it's set to "Rejected". This is a key part of automating the enrolment workflow and ensures the record reflects the latest state of the approval lifecycle.

8. Tasks

Although not implemented yet, **Tasks** can be auto-created using Flows to assign follow-up actions. For example, if a student's GPA drops below 2.0, a task could be assigned to a counsellor to intervene. Tasks help ensure accountability and follow-through on critical actions without manual assignment.

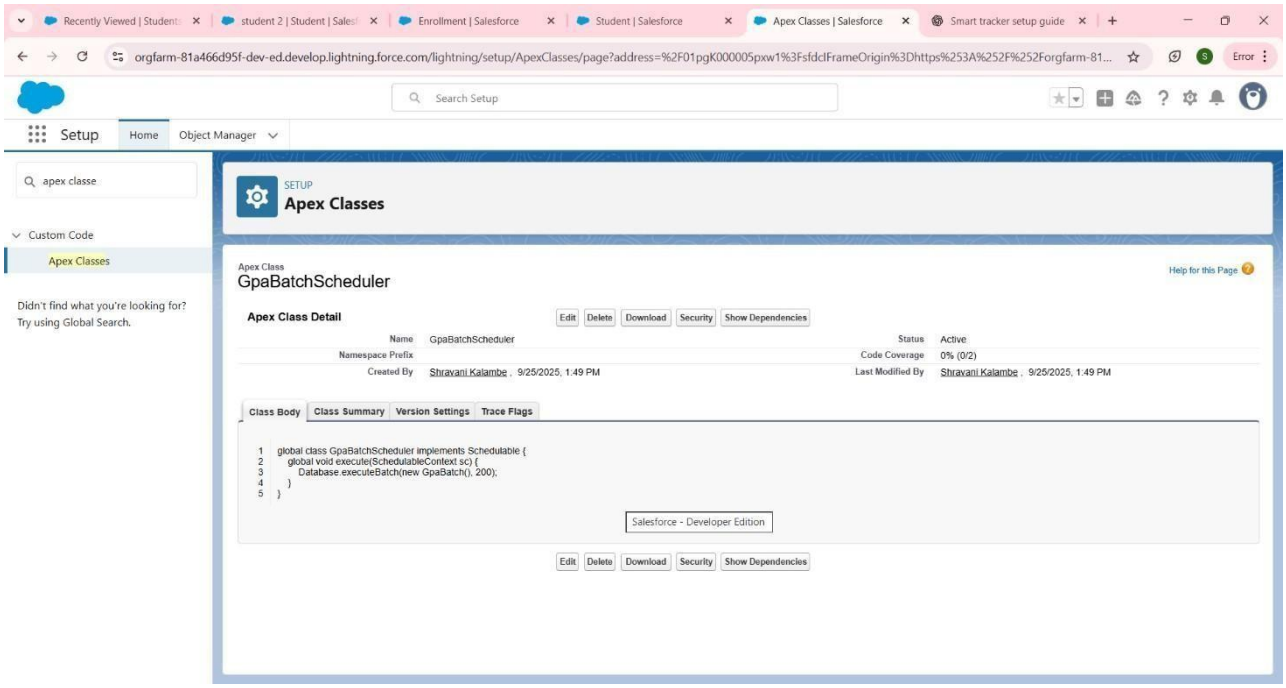
9. Custom Notifications

Custom Notifications can be configured via **Flows** to notify users inside the Salesforce UI or mobile app. We can notify students or faculty when key events occur — like successful enrollment or when GPA triggers a counselling flag. These are more modern than email and appear directly in the Salesforce bell icon.

Phase 5: Apex Programming (Developer)

1. Classes & Objects

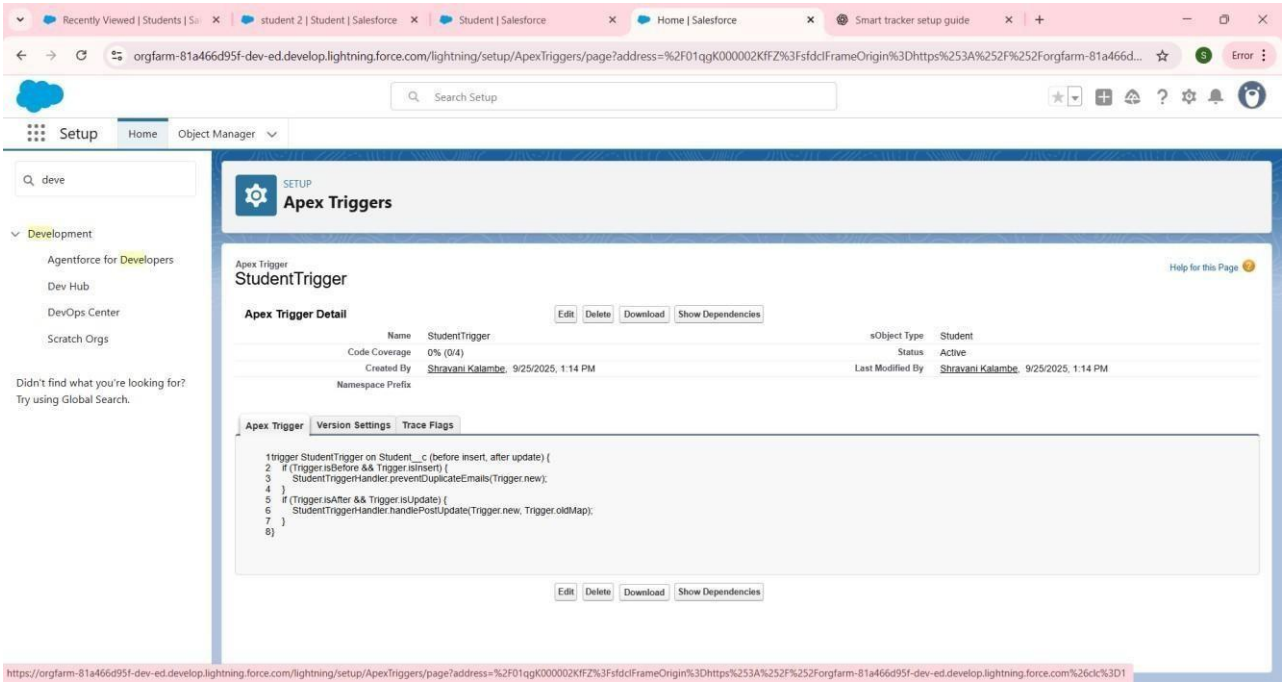
We created **Apex classes** like GpaBatch and GpaScheduler. These classes encapsulate logic in methods, and objects (like Student__c, Enrolment__c) are manipulated inside them. This shows how Salesforce data (custom objects) interact with code structures (Apex classes).



2. Apex Triggers (before/after insert/update/delete)

We built **before insert** and **before update** triggers on Student__c.

- Before insert → blocks duplicate emails.
- Before update → flags counseling and publishes a platform event if GPA < 2.0. This shows

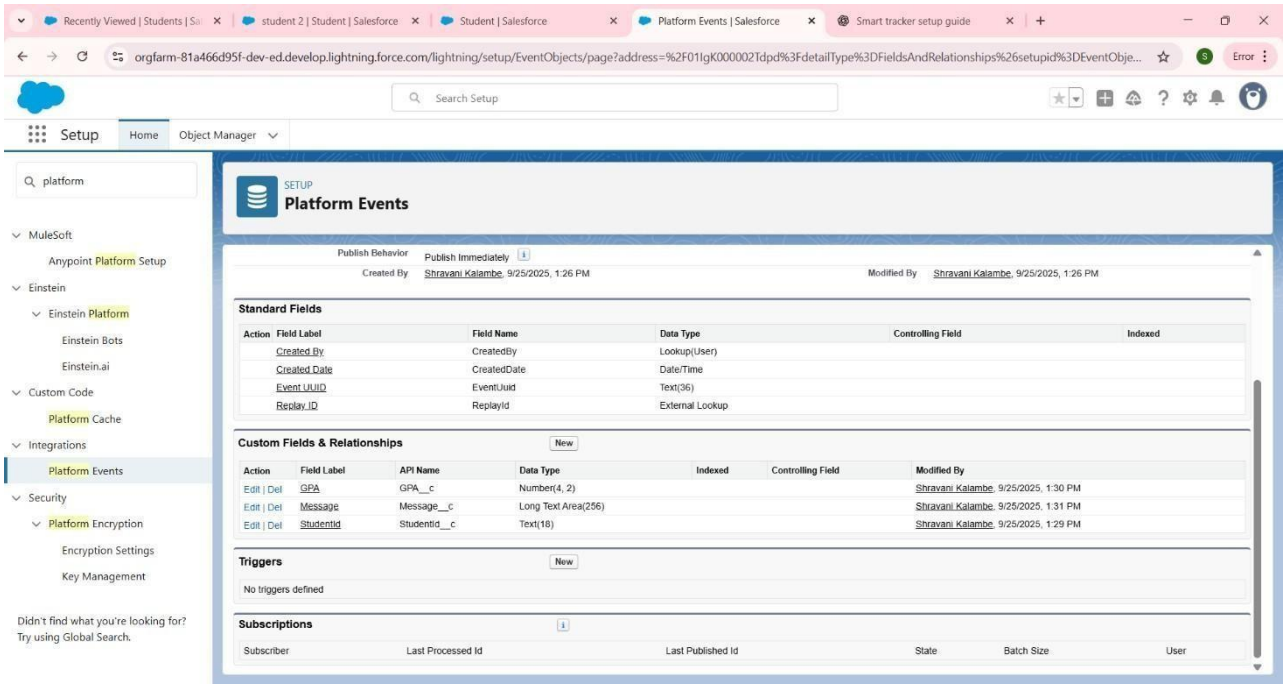


how triggers can enforce rules before saving data.

3. Trigger Design Pattern
We separated logic into **helper classes** instead of writing heavy logic directly in triggers. This is a best practice called the **Trigger Handler Pattern**, which makes code reusable, easier to test, and maintainable.

4. SOQL & SOSL
We used **SOQL queries** to fetch existing students (for duplicate email check) and to calculate GPA from Enrolment__c. Aggregate queries with AVG() were applied in the batch. SOSL wasn't needed here but is useful for text searches across objects.

5.Platform Event (LowGPAAlert__e)
The **LowGPAAlert__e event** is a custom platform event used to broadcast low GPA cases. When the GPA trigger fires, this event is published with student details. Other Salesforce processes or external apps can **subscribe** to this event to take actions like sending notifications or assigning tasks.



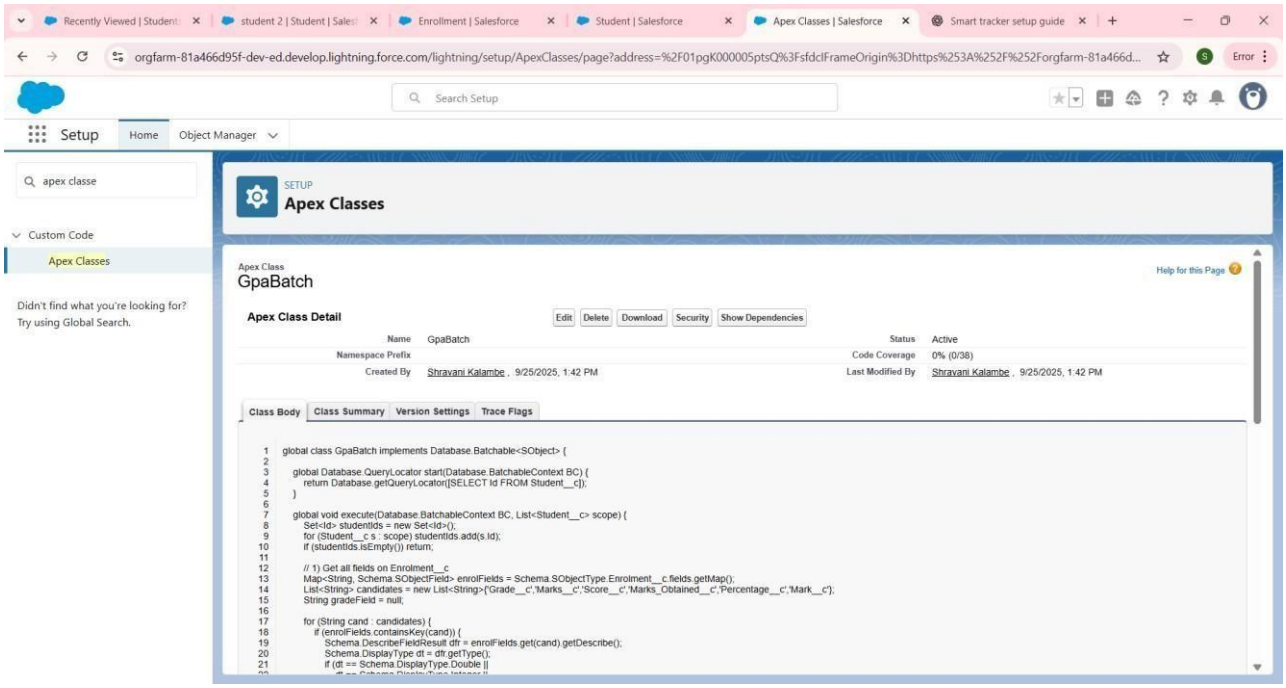
6. Collections: List, Set, Map

- **List**: stored students to update.
- **Set**: used for unique student IDs in batch scope.
- **Map**: stored StudentId → GPA pairs from aggregate queries.
  Collections improved performance and avoided duplicates in processing.

7. Control Statements
We used **if conditions and loops** to check duplicates, validate GPA, and loop through batch records. For example, if(avgMap.containsKey(s.Id)) ensures only students with enrolments are updated.

8. Batch Apex

We created GpaBatch to recalculate GPA in bulk. Batch Apex processes records in chunks, avoiding governor limits. The execute() method handled GPA recalculation and record updates efficiently.

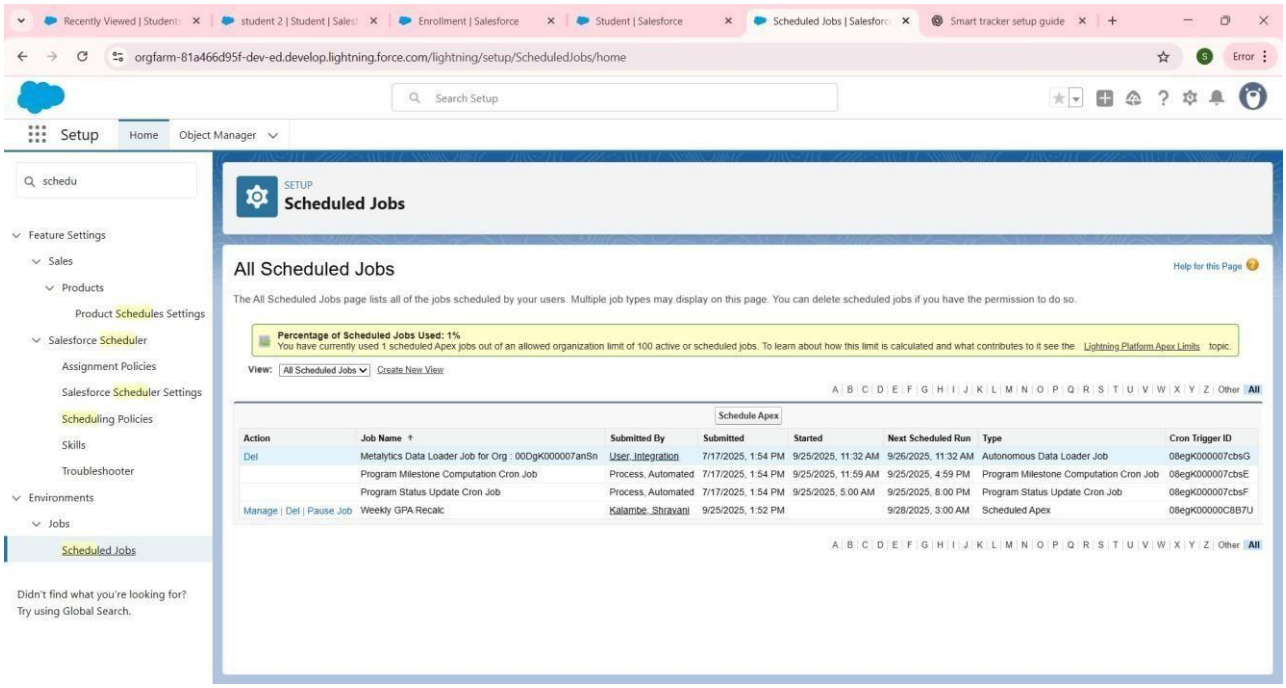


9. Queueable Apex

(Not yet built, but could be added.) Queueable Apex is a lightweight async process. Example: publishing Low GPA alerts to external systems using System.enqueueJob() would be a Queueable use case.

10. Scheduled Apex

We implemented GpaScheduler to run the GPA Batch every Sunday at 3 AM. Scheduled Apex uses CRON expressions and helps automate repetitive jobs. This ensures GPA data stays fresh without manual runs.



10. Future Methods

(Not yet built, but relevant.) Future methods are used for asynchronous operations like sending callouts or heavy processing. For example, emailing students when GPA is recalculated could be done in a @future method.

11. Exception Handling

We used **try–catch blocks** in the LWC Apex controller (StudentController) to handle errors and return messages. Exception handling ensures the system doesn't crash and users see friendly error messages.

Phase 6: User Interface Development

1. Lightning App Builder

We used **Lightning App Builder** to edit the **Student record page**. We dragged our custom LWC (student Progress) onto the layout.
Activated the page for the **Student Management App** so it shows to Faculty/Admin. Result: Users see the custom component when opening any Student record.

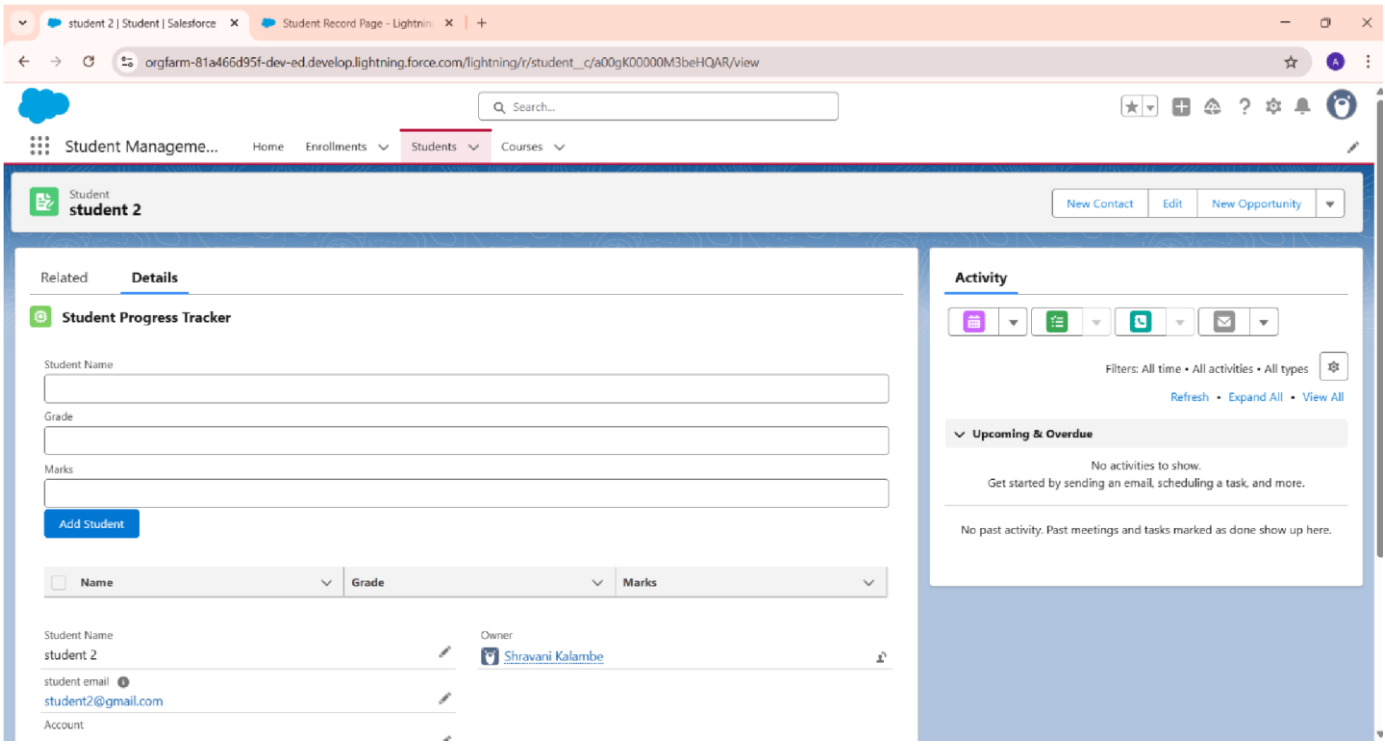
2. Record Pages

The Student object's **Record Page** was customized.

Default tabs (Details, Related) were kept as is, and the custom component was placed on the page.
This ensures the Student record view is tailored with additional functionality (progress bar + button).
Result: Record page now has both standard fields and the custom LWC.

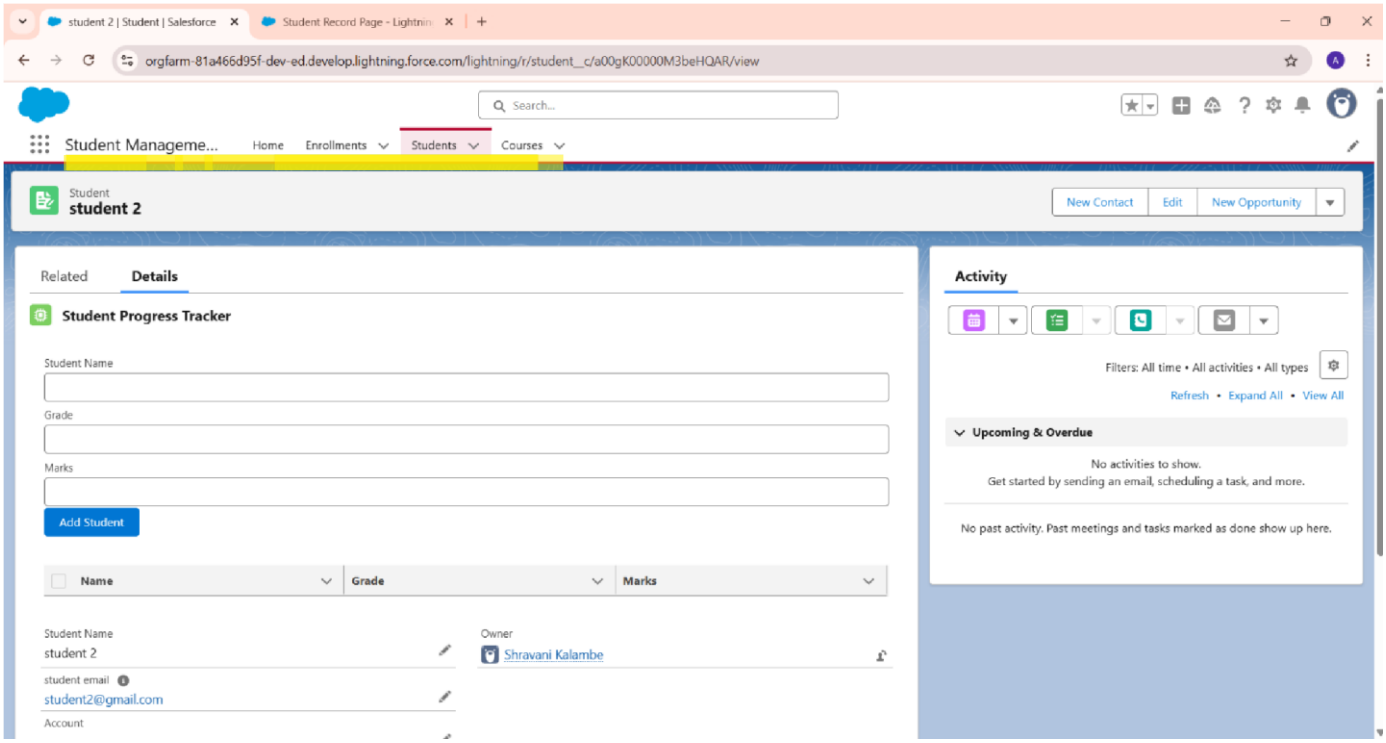
3. Tabs

In **App Manager**, we edited the Student Management app navigation. Added **Students, Courses, Enrollments** as navigation items.



This gives Faculty/Admin direct access from the top navigation bar. Result: App navigation is organized and user-friendly.

4. Home Page Layouts



❌ **Not Done / Not Required** for your project.
We didn't customize the Lightning Home Page layout (welcome screen/dashboard). Result: Home Page remains default.

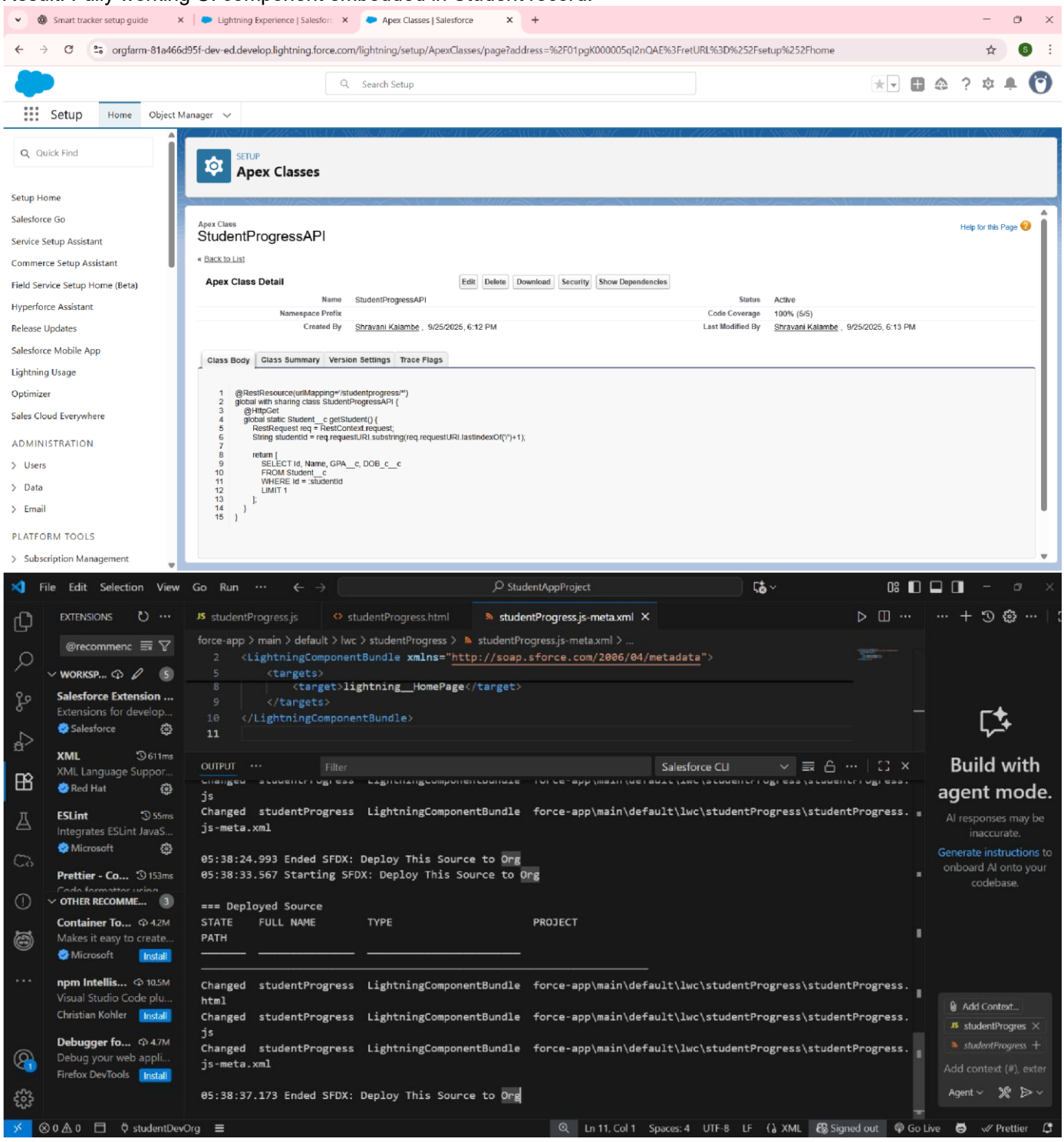
5. Utility Bar

❌ **Not Done / Optional**.
We did not add any **Utility Bar items** (like Notes, History, or custom components). Result: Utility Bar is not part of Phase 6 scope.

6. LWC (Lightning Web Components)

Created **studentProgress LWC**.
Includes GPA display, progress bar, and button for "Request Re-evaluation." Configured with an XML file so it's exposed only on **Student__c Record Pages**.

Result: Fully working UI component embedded in Student record.



7. Apex with LWC

Built StudentController.cls Apex class.

Connected it to the LWC via **imperative Apex call** (button → Apex method).

Handles the re-evaluation request and returns success/error messages. Result: UI → Server → UI round-trip working properly.

8. Events in LWC

❌ **Not Done / Not Required**.
We didn't use custom events between components (e.g., child → parent communication).
Only **standard ShowToastEvent** was used to show notifications. Result: Events beyond toast handling not needed for now.

9. Wire Adapters

Used @wire(getRecord, { recordId: '$recordId', fields: [GPA_FIELD] }).
Automatically fetches the **GPA__c** field for the Student record.
Displayed value in UI and linked it to progress bar dynamically. Result: Data loads reactively without manual calls.

10. Imperative Apex Calls

Button click calls requestReevaluation({ studentId: this.recordId }).
Shows toast message for success/failure.
Imperative Apex is triggered only on user action (button). Result: On-demand server interaction built successfully.

11. Navigation Service

❌ **Not Done / Not Required**.
We did not implement **NavigationMixin** (e.g., navigating to another record, page, or app). Result: No navigation actions configured in LWC.

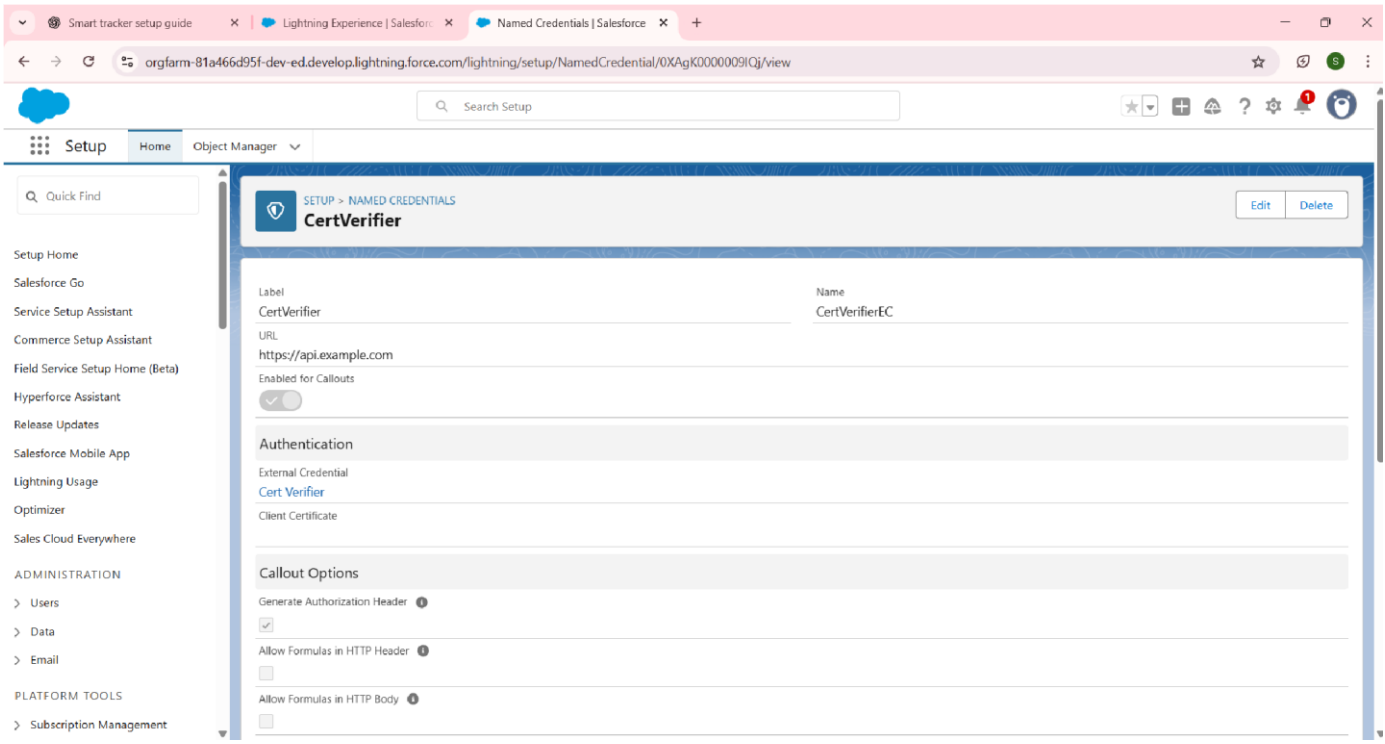
Summary

- **Done:** Lightning App Builder, Record Pages, Tabs, LWC, Apex with LWC, Wire Adapters, Imperative Apex Calls.
- **Not Applicable / Skipped:** Home Page Layouts, Utility Bar, Events in LWC (custom ones), Navigation Service.

Phase 7 – Integration & Events

1. Named Credentials

We created a **Named Credential** called **CertVerifier** to securely store the base URL of an external API for certificate verification.
This lets Apex use the alias `callout:CertVerifier` instead of hard-coding full URLs or authentication details.
For testing, we used a public API (JSONPlaceholder) with **No Authentication**. This improves security and makes callouts easier to maintain.



2. Remote Site Settings

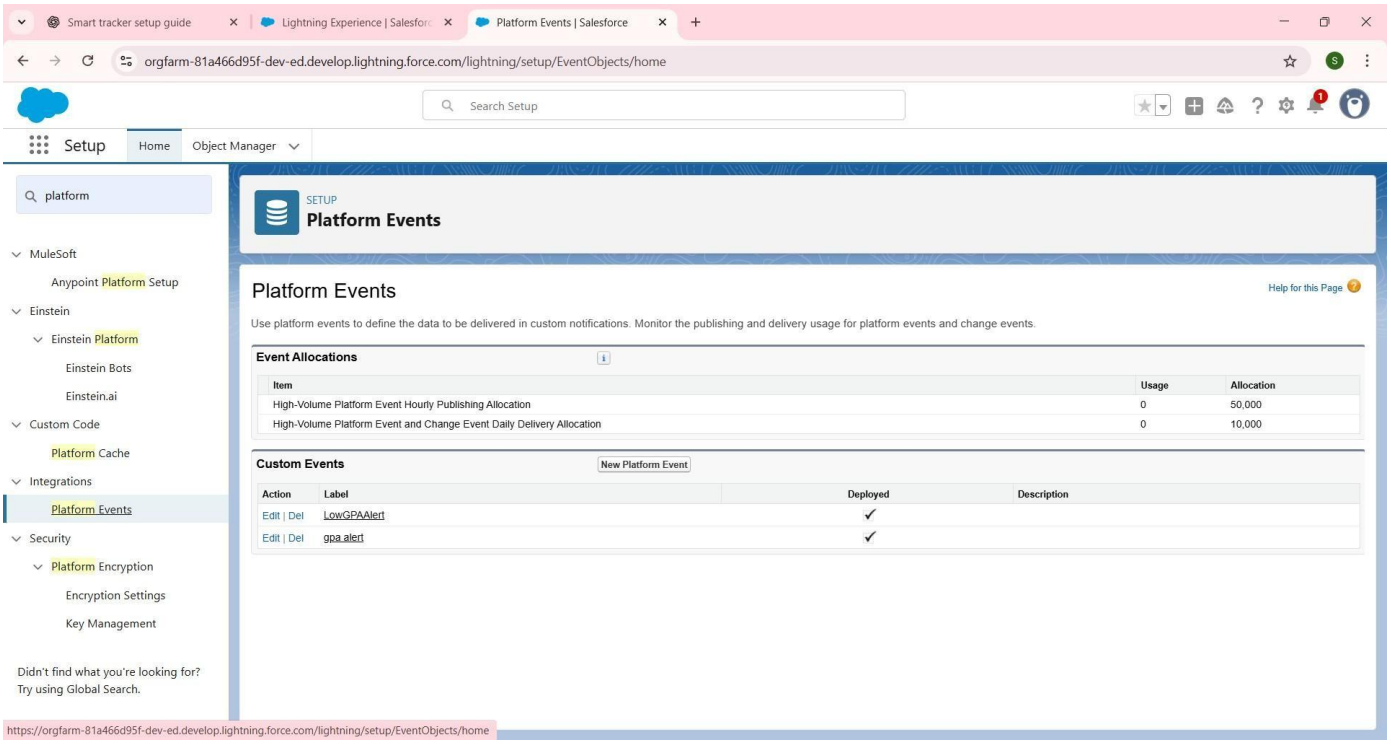
As a fallback for environments without Named Credentials, we also created a **Remote Site Setting** with the same API base URL.
This step ensures Salesforce can connect to external endpoints when Named Credentials are not used. In our demo, we relied on the Named Credential, but the Remote Site is a backup to avoid callout-blocking errors.

3. Web Services (REST) & Callouts

We implemented a simple Apex class **CertVerifierService** to perform a REST callout to the external API using the Named Credential.
The class sends an HTTP GET request and returns the API's response.
We tested the callout in **Developer Console → Execute Anonymous** and confirmed we could retrieve live JSON data.
This proves that our Salesforce app can communicate with external services.

4. Platform Events

We created a **Platform Event** named **LowGPAAlert__e** with fields for **StudentId**, **GPA**, and **Message**.
Whenever a student's GPA drops below 2.0, our existing **Student Trigger** publishes this event. This allows downstream systems or even Salesforce triggers to react in near real-time to academic performance issues.
It decouples internal logic from external listeners.



5. Change Data Capture (CDC)

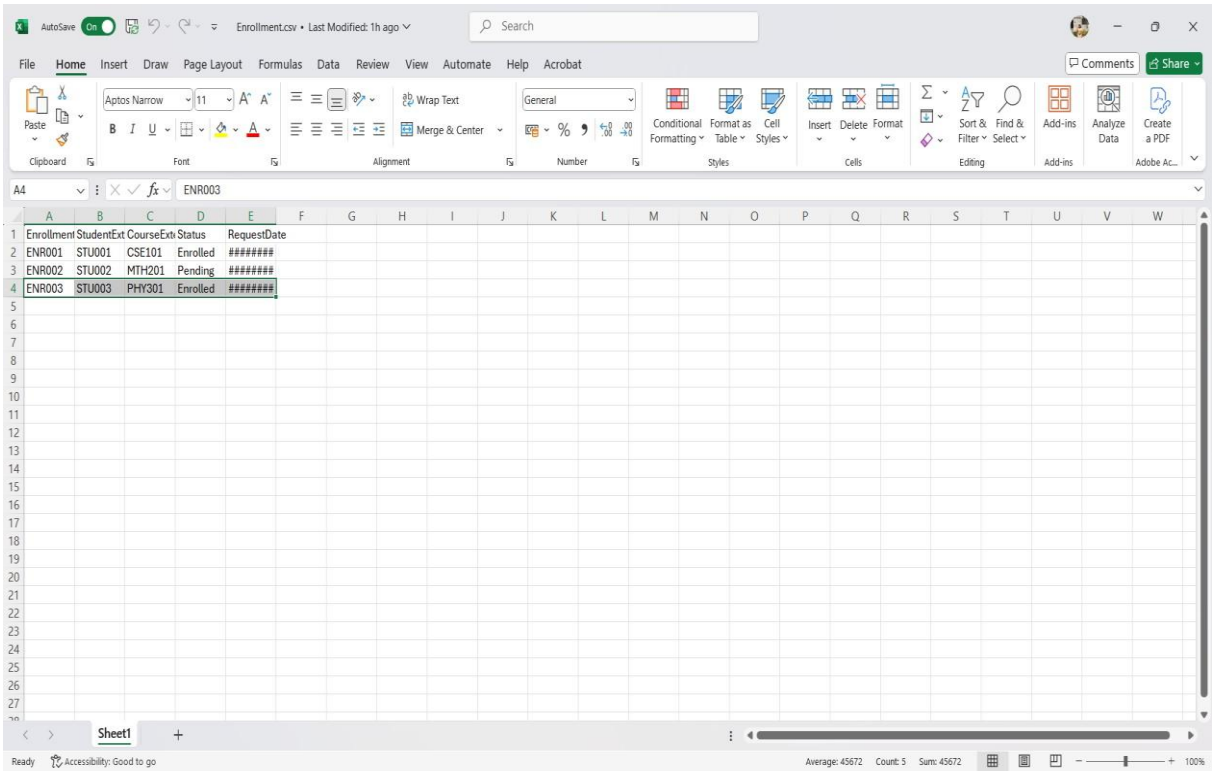
We enabled **Change Data Capture** for the **Student__c** and **Enrollment__c** objects.
This feature automatically publishes record-change events whenever these objects are created, updated, or deleted.
We tested it by subscribing to `/data/StudentChangeEvent` in **Workbench** and verified that updates in Salesforce trigger real-time events.
CDC ensures external systems stay synchronized without manual exports.

End Result:
Phase 7 equips our app with secure external integration via Named Credentials and REST callouts, plus real-time messaging through **Platform Events** and **CDC**, enabling smooth two-way communication with external systems and better monitoring of student progress.

Phase 8: Data Management & Deployment

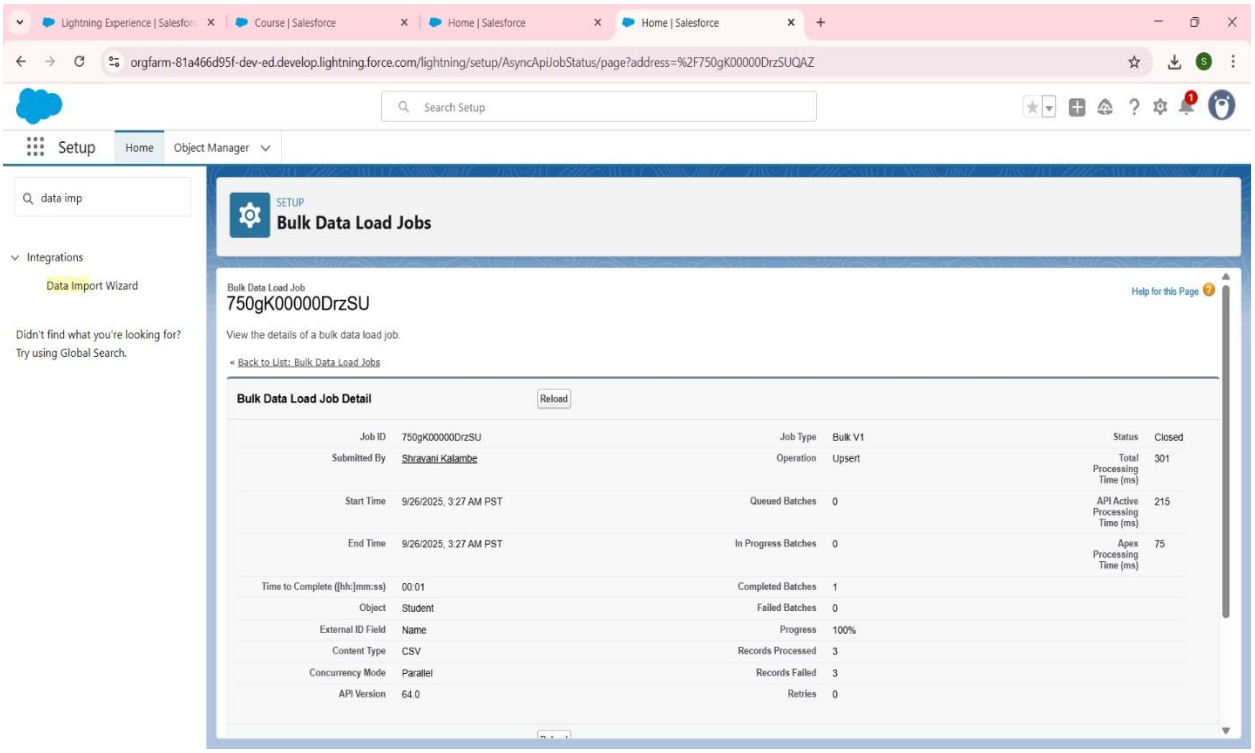
1) Prepare CSVs

Create three CSVs: **Students, Courses, Enrollments**.
Each should have an **External ID column** (unique text field).
Ensure clean data: no formulas, saved as UTF-8.
Verify headers and sample rows in a text editor before upload.



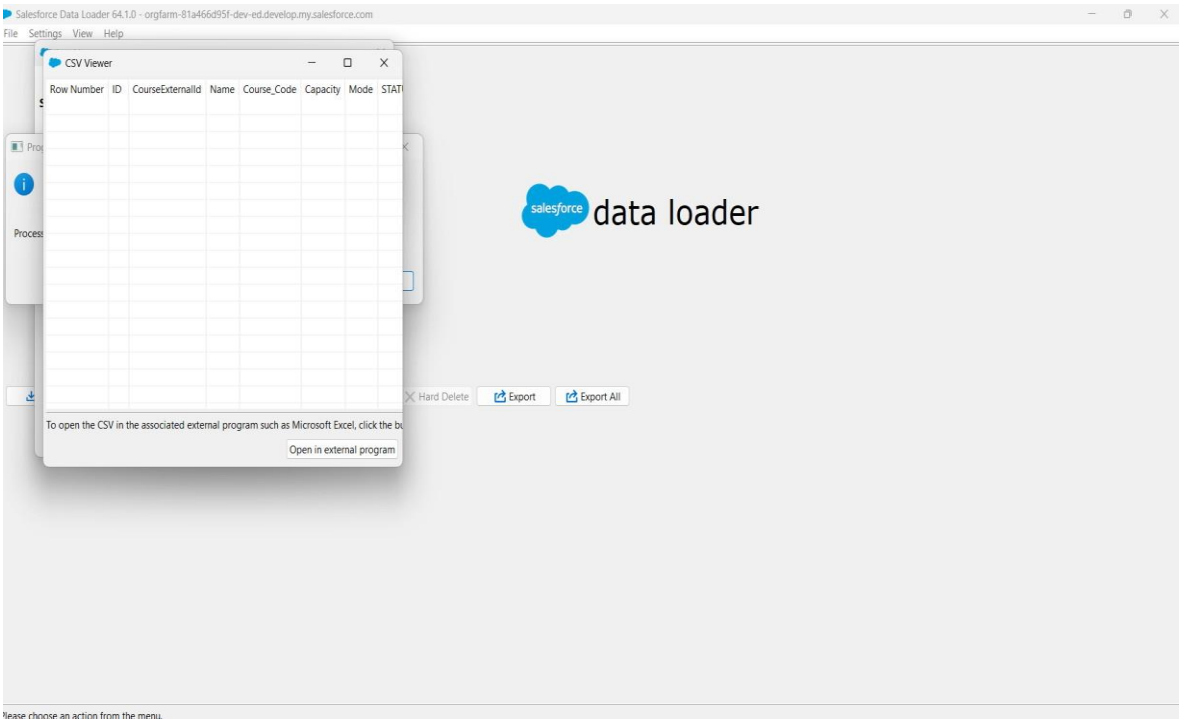
2) Data Import Wizard

Use for **small/simple loads** (students, courses).
Setup → Data Import Wizard → choose object.
Upload CSV, map headers → Salesforce fields.
Run import, check email + sample records.



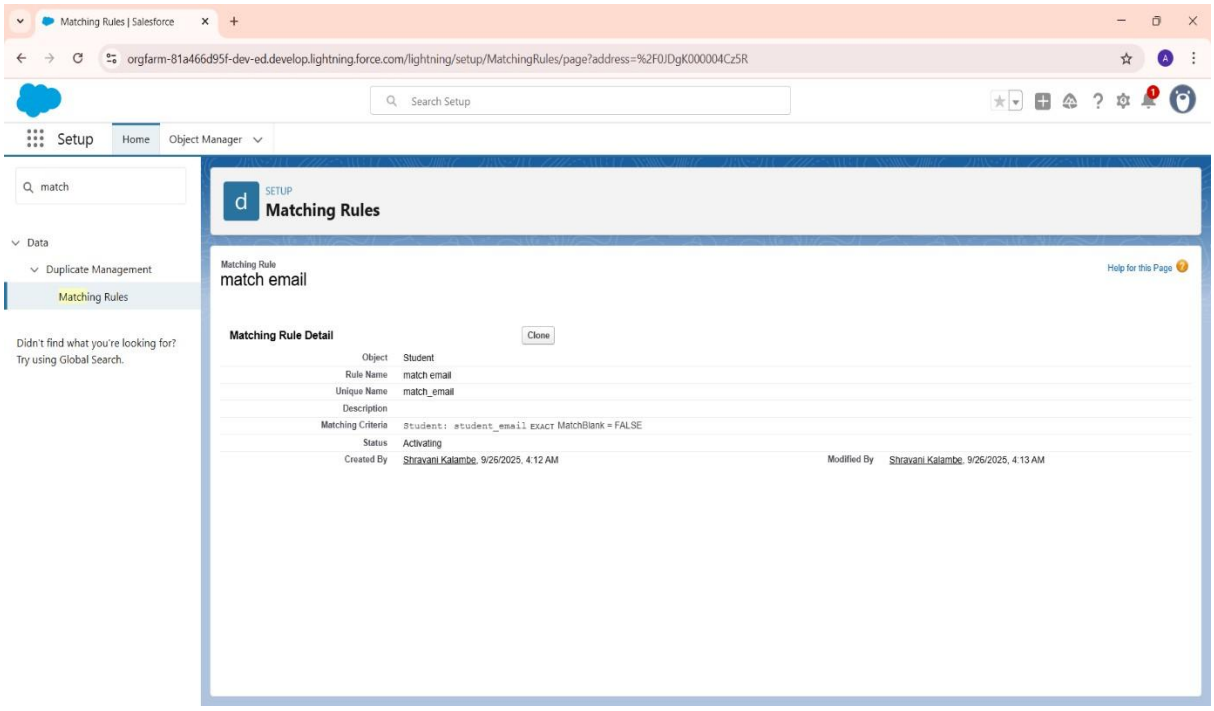
3) Data Loader

Best for **large or complex loads** (like enrollments with lookups).
Download & install from Setup → Data Loader.
Use **Insert/Update/Upsert** with CSV and mapping.
Review success & error CSVs after run.



4) Duplicate & Matching Rules

Prevent duplicate Students in the system.
Create **Matching Rule** (e.g., match Email or Name+DOB).
Create **Duplicate Rule** to block or alert on duplicates.
Test by inserting a duplicate → confirm block/alert works.



5) Data Export & Backup

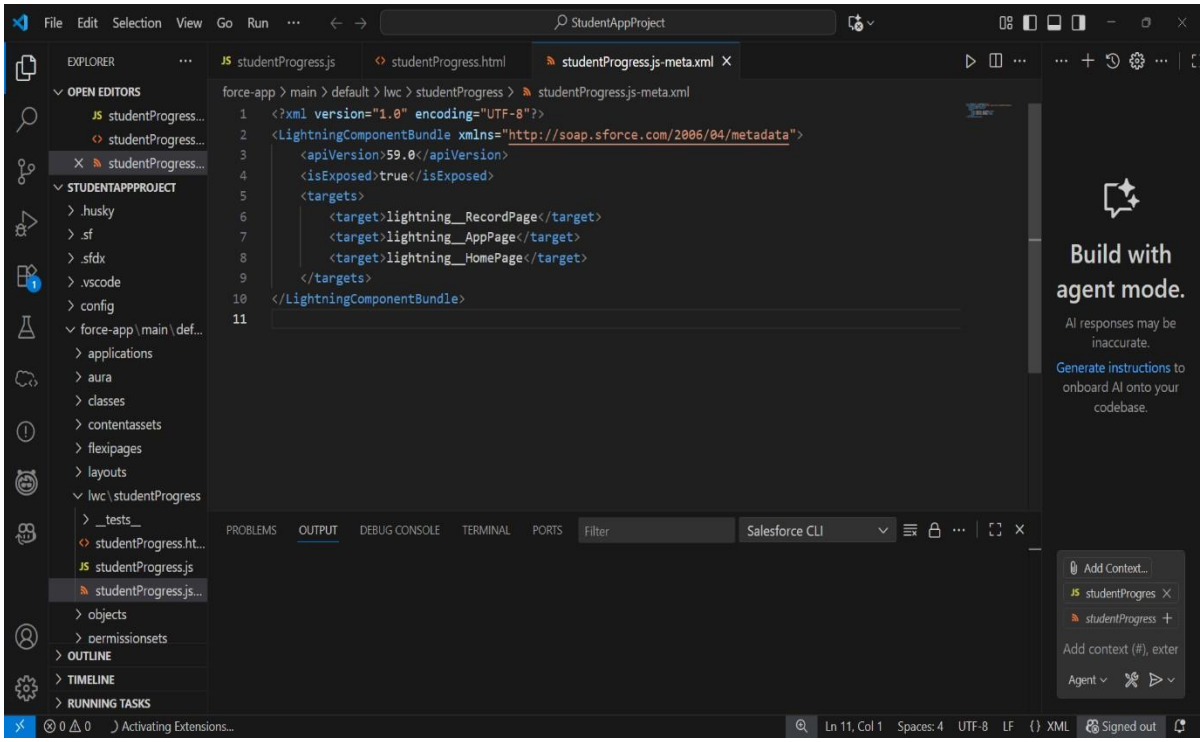
Always back up before big imports/changes.
Setup → Data Export → choose objects → Export Now.
Download and unzip CSV package securely.
Schedule weekly exports for safety.

6) Change Sets

Use to move metadata Sandbox → Production.
Create **Outbound Change Set** → add components (objects, fields, Apex, LWC).
Upload, then deploy in **Inbound Change Sets** in target org.
Validate first, then deploy and test.

7) VS Code & CLI
Best for source-driven development.
Install VS Code + Salesforce CLI, create a project.

Retrieve metadata (sf project retrieve) and deploy (sf project deploy). Use Git for version control, test before deploying.



8) Unmanaged vs Managed Packages

Unmanaged: one-time copy of components (internal use).
Managed: with namespace + versioning (AppExchange).
For internal projects, mainly use Change Sets or CLI.
Packages are optional, not required.

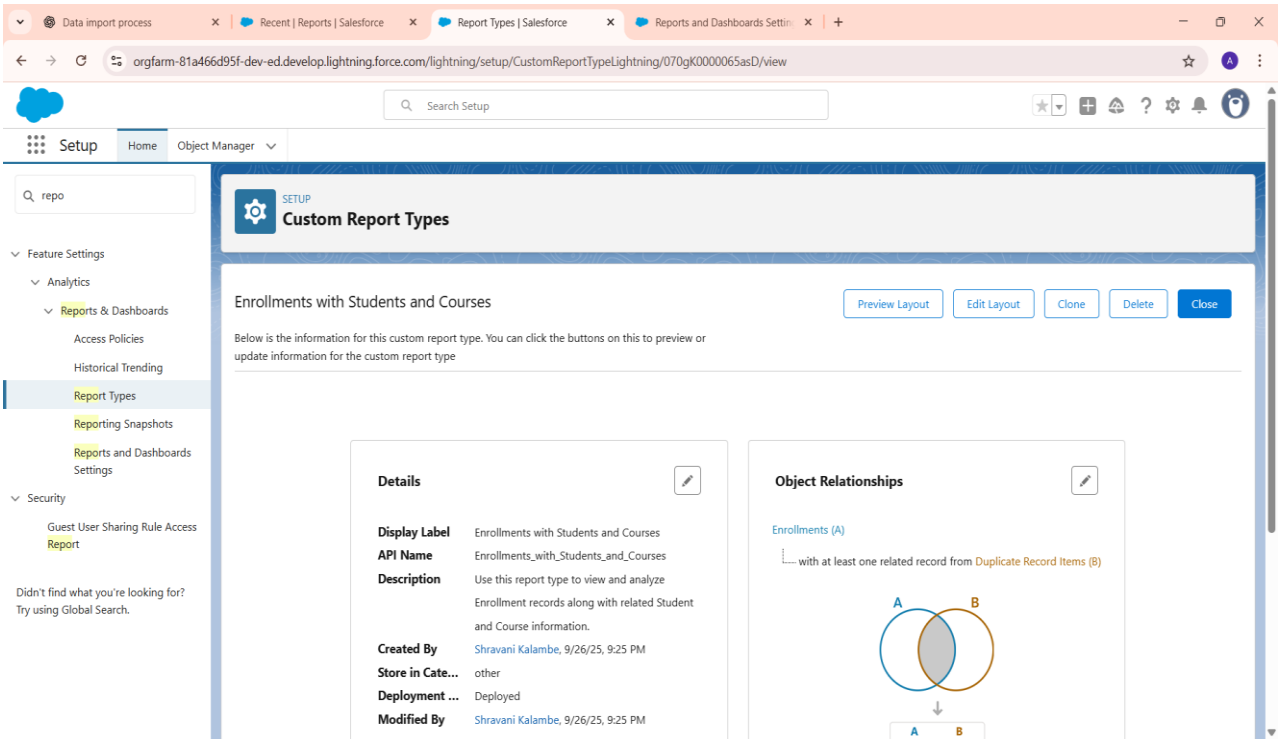
Phase 9: Reporting, Dashboards & Security Review

Reports

Reports let you analyze Salesforce data in different formats: Tabular (simple lists), Summary (grouped rows), Matrix (rows & columns), and Joined (combine multiple blocks). You build them in the Report Builder by selecting fields, filters, and groupings, then save and share.

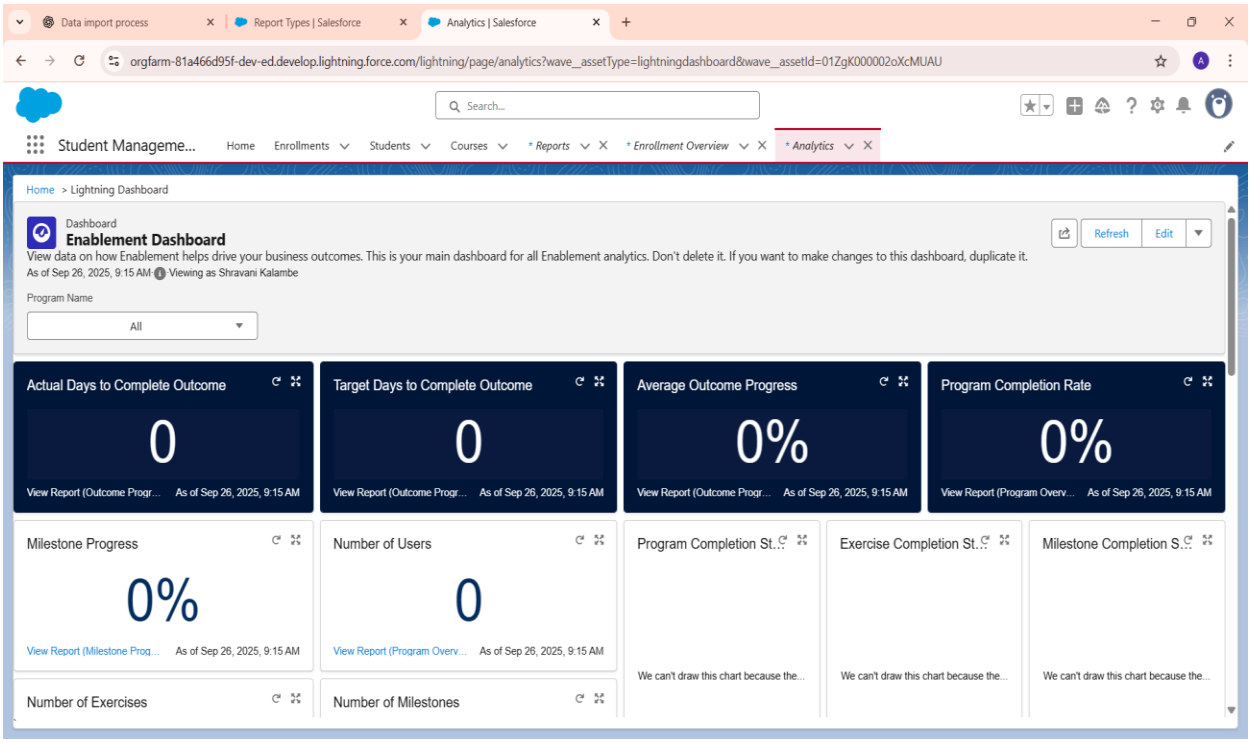
Report Types

A Report Type defines which objects and fields are available when building a report. You select a Primary Object (like Students or Enrollments), then add related objects to expose more data. Custom report types let you tailor reporting to your org's data model.



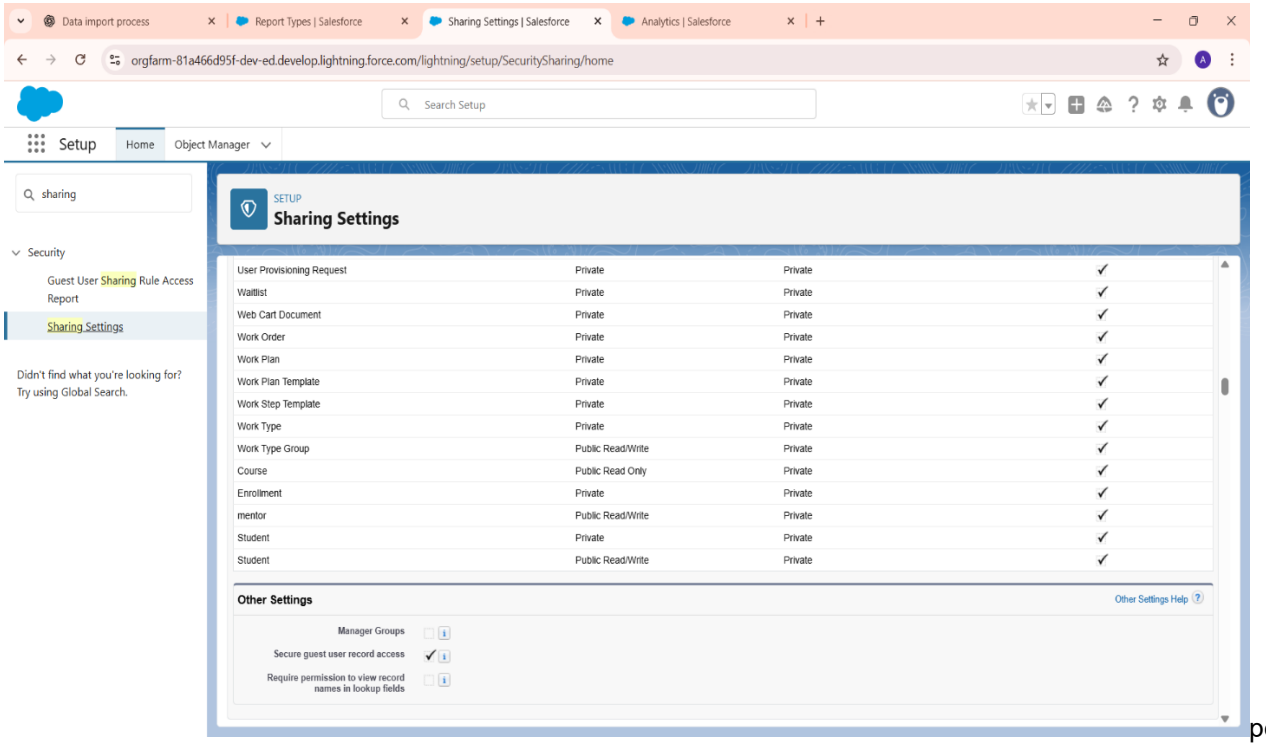
Dashboards
Dashboards visualize reports with charts, metrics, gauges, and tables in one view. Each component is powered by a saved report and can be resized, arranged, and filtered. They give managers and users an at-a-glance view of key business indicators.

Dynamic Dashboards
Dynamic Dashboards show data based on the logged-in user's security and sharing settings. This ensures each person only sees the records they are allowed to access. They are especially useful for role-based organizations with varying data visibility needs.



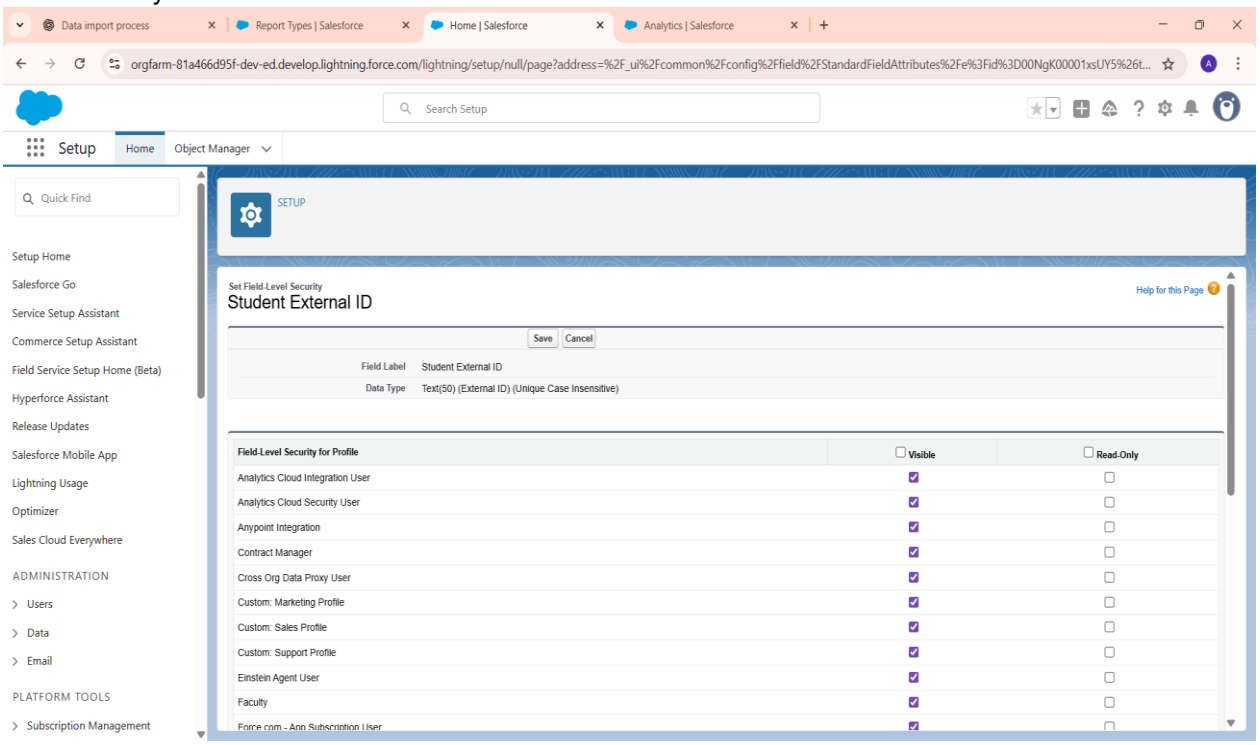
Sharing Settings

Sharing Settings control record visibility across users. You set Org-Wide Defaults (OWD) first, then open up access with Roles, Public Groups, and Sharing Rules. This ensures data is kept private by default but shared with the right

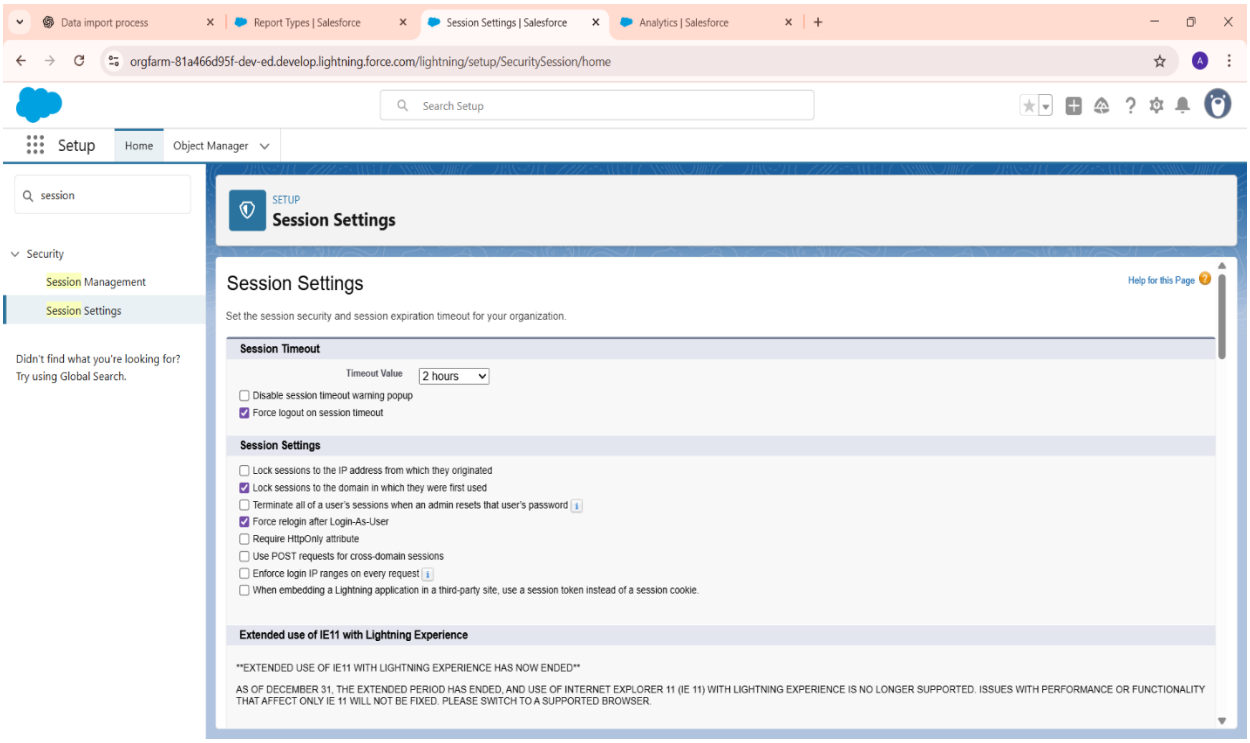


people.

Field-Level Security (FLS)
FLS restricts access to individual fields on objects. You can make fields visible, read-only, or hidden for specific profiles or permission sets. This protects sensitive information, like emails or personal IDs, from being exposed unnecessarily.



Session Settings
Session Settings define how long users stay logged in and how sessions are secured. You can set timeouts, lock sessions to IP addresses, and control persistent logins. These settings reduce the risk of unauthorized access if devices are left unattended.



Login IP Ranges

Login IP Ranges restrict where users can log in from. At the org level, trusted IPs bypass security tokens; at the profile level, strict ranges ensure logins only come from approved networks. This adds an extra layer of security, especially for remote access.

Audit Trail

Audit Trail helps track system and data changes. Setup Audit Trail records configuration changes (who, when, what), while Field History Tracking logs changes to selected fields. This supports compliance, accountability, and troubleshooting by showing change history.