**Salesforce Project Report: QuickServe Management System**

**An Automated Order & Loyalty CRM Implementation**

**♦️ Project Overview:**

The QuickServe Management System is a comprehensive Customer Relationship Management (CRM) solution built on the Salesforce platform, designed specifically for a quick-service restaurant. The system's core function is to replace manual order tracking and implement a fully automated customer loyalty program. Key features include custom objects for managing orders, reservations, and loyalty accounts; a real-time, automated process for calculating and awarding loyalty points upon order completion; and a custom user interface for staff to view a 360-degree profile of each customer, including their order history and current points balance. The CRM also features an integration endpoint to receive orders from an external Point-of-Sale (POS) system and provides managers with reports and dashboards to monitor customer engagement and business performance.

**♦️ Objectives:**

The primary objective of this project is to centralize all customer and transactional data within a single, scalable Salesforce environment to drive customer retention and streamline operations. By automating the loyalty points program, the CRM aims to eliminate manual data entry, ensure real-time accuracy, and enhance the customer experience, thereby increasing repeat business. A further goal is to empower restaurant staff with an intuitive interface for better customer management at the point of sale. For management, the objective is to provide actionable insights into business performance through comprehensive reports and dashboards, linking customer loyalty directly to business value and enabling data-driven decision-making.

**Phase 1: Problem Understanding & Industry Analysis**

**♦️ Requirement Gathering**

**Functional Requirements**

* The system must automatically calculate loyalty points when an order's status is set to 'Complete'.
* Staff should be able to view a customer's order history and current points balance.
* Managers need a feature to manually adjust points for promotions or customer service reasons.
* The system should be able to track orders, including menu items, quantity, and total amount.
* Notifications (e.g., email) should be configurable for key events like order confirmation.

**Non-Functional Requirements**

* **Data Security:** Only authorized users (staff, managers) should be able to access or modify customer and order data.
* **Performance:** The points calculation and record updates should happen in near real-time without slowing down the user experience.
* **Usability:** The interface for viewing points and order history should be simple and intuitive for restaurant staff.
* **Scalability:** The system must be able to handle a large volume of daily orders and customer records without degradation in performance.

**♦️ Stakeholder Analysis**

| Stakeholder | Role in the System | Needs/Expectations |
| --- | --- | --- |
| **Restaurant Staff** | End users who manage orders and interact with customers. | Quick access to customer points and order history. |
| **Managers** | Oversee operations and customer satisfaction. | Visibility into loyalty trends, ability to resolve issues, and performance reports. |
| **Customers** | Indirect users who benefit from the loyalty program. | Accurate and timely updates to their loyalty points. |
| **Admin** | Manages the Salesforce platform and configuration. | Control over the automation rules, data, and user permissions. |

**♦️ Business Process Mapping**

* **Step 1:** A staff member creates or updates an Order\_\_c record for a customer in Salesforce.
* **Step 2:** The staff member updates the order's status to 'Complete' once the transaction is finished.
* **Step 3:** An automated process (Apex trigger) fires upon the status change.
* **Step 4:** The system calculates loyalty points based on the order's total amount.
* **Step 5:** The calculated points are automatically added to the customer's Loyalty\_Program\_\_c record.
* **Step 6:** The customer's new points balance is immediately visible to staff on the Contact page.
* **Step 7:** Managers can view reports on points awarded and customer loyalty trends.

**♦️ Industry-Specific Use Case Analysis**

* **QSR/Food & Beverage Industry:** This solution provides a direct way to increase customer retention and engagement through an automated loyalty program, a common strategy in this competitive market.
* **Real Estate Industry:** Property owners get a new market (students) without brokers.

*(Note: This is from the template and less relevant here, but included for structural consistency).*

* **Technology/CRM Industry:** Salesforce serves as a powerful, scalable platform for managing the entire customer lifecycle, from order-taking to loyalty and retention, all in one place.

**♦️ AppExchange Exploration**

Before building a custom solution, we explored the Salesforce AppExchange for existing loyalty management packages:

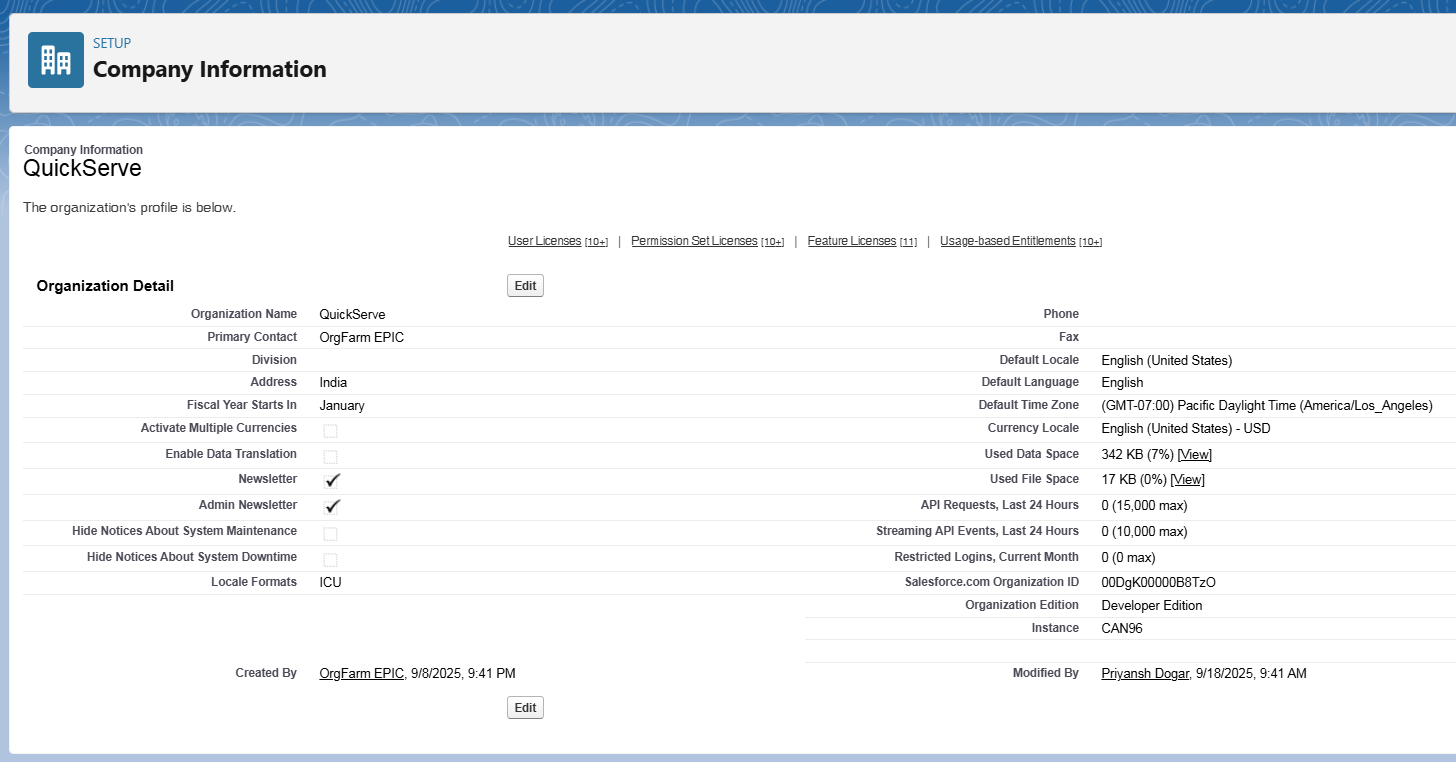
* **Large-Scale Loyalty Apps:** Many are designed for enterprise-level, multi-channel marketing campaigns and are too complex and expensive for this specific use case.
* **Order Management Apps:** These focus heavily on inventory and fulfillment rather than the post-purchase loyalty calculation.
* **Point-of-Sale (POS) Apps:** While some have loyalty features, they often don't integrate as deeply or flexibly into the Salesforce Contact record as a custom solution can.

**PHASE 2 - Org Setup & Configuration**

**♦️ Company Profile Setup**

Basic org details are configured under **Setup → Company Information → Edit** to establish the foundational settings for the QuickServe org.

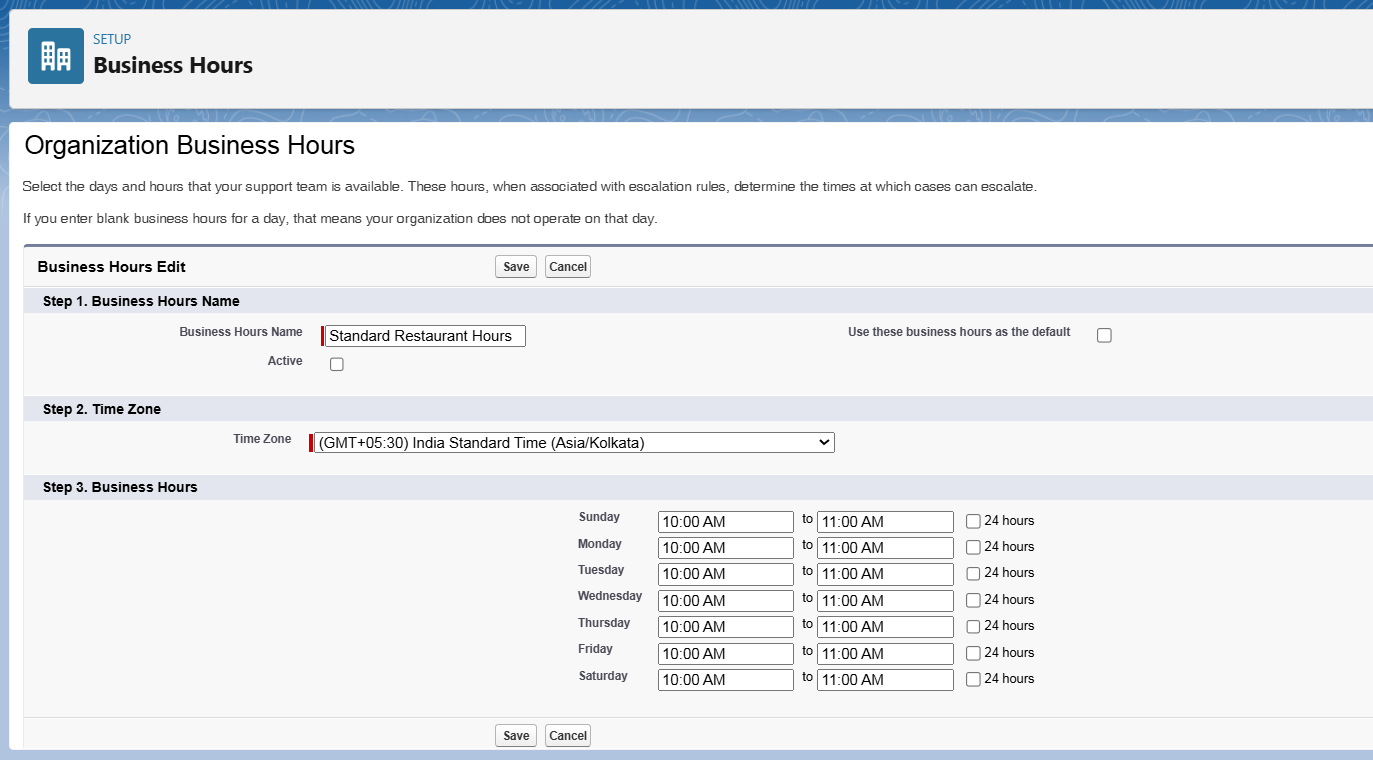
* **Name**: QuickServe Management System
* **Time Zone**: GMT+05:30 Asia/Kolkata
* **Locale**: English (India)
* **Language**: English
* **Currency**: INR



**♦️ Business Hours Setup**

Working hours are configured to reflect restaurant operating times for future case management and SLA tracking.

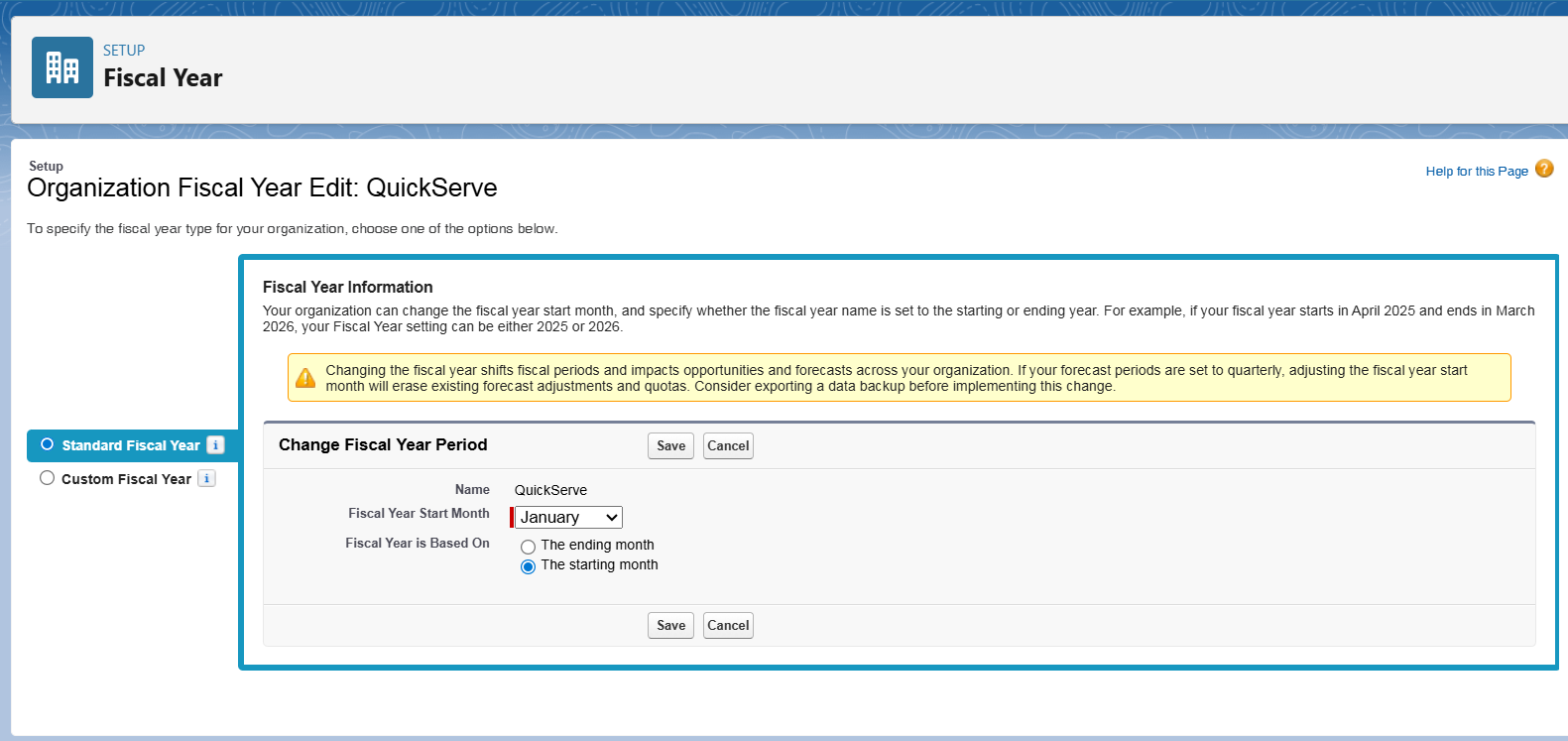
* **Path**: Setup → Business Hours → New.
* **Name**: Standard Restaurant Hours.
* **Time Zone**: GMT+05:30 Asia/Kolkata.
* **Working Hours**: Mon-Sun 10:00 AM–10:00 PM.



**♦️ Fiscal Year Setup**

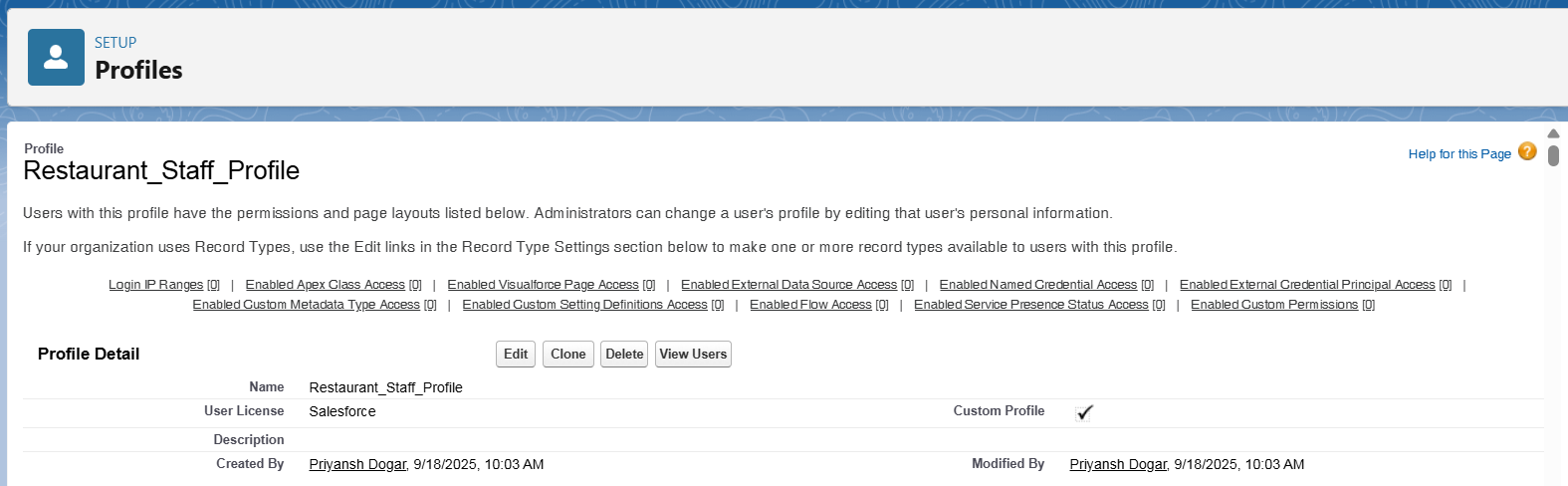
A standard fiscal year is established to define reporting periods for sales and order analysis.

* **Path**: Setup → Fiscal Year.
* **Type**: Standard Fiscal Year.
* **Configuration**: The starting month is set to **January**.

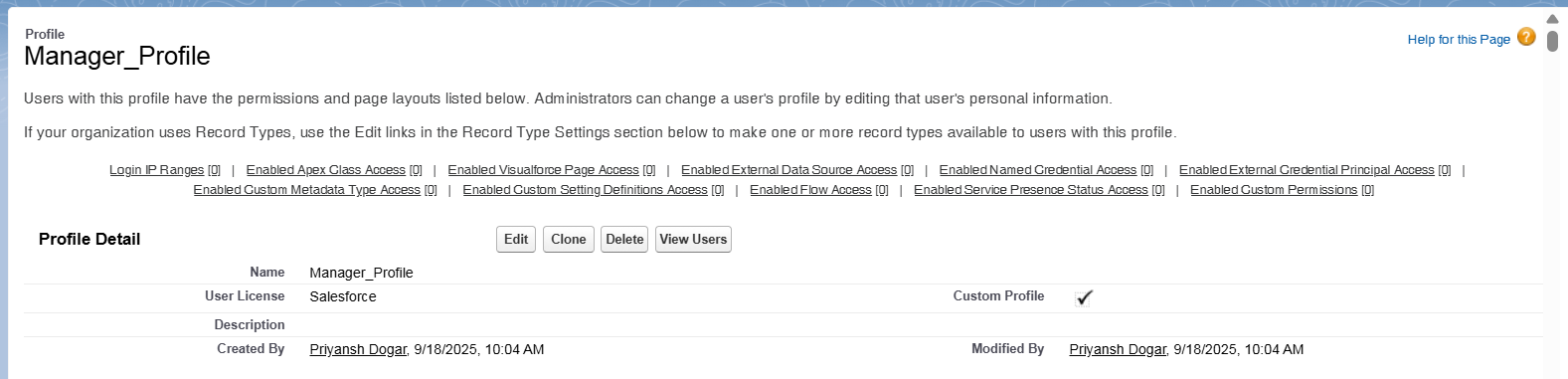


**♦️ User Setup (Profiles, Roles, Permission Sets, Users)**

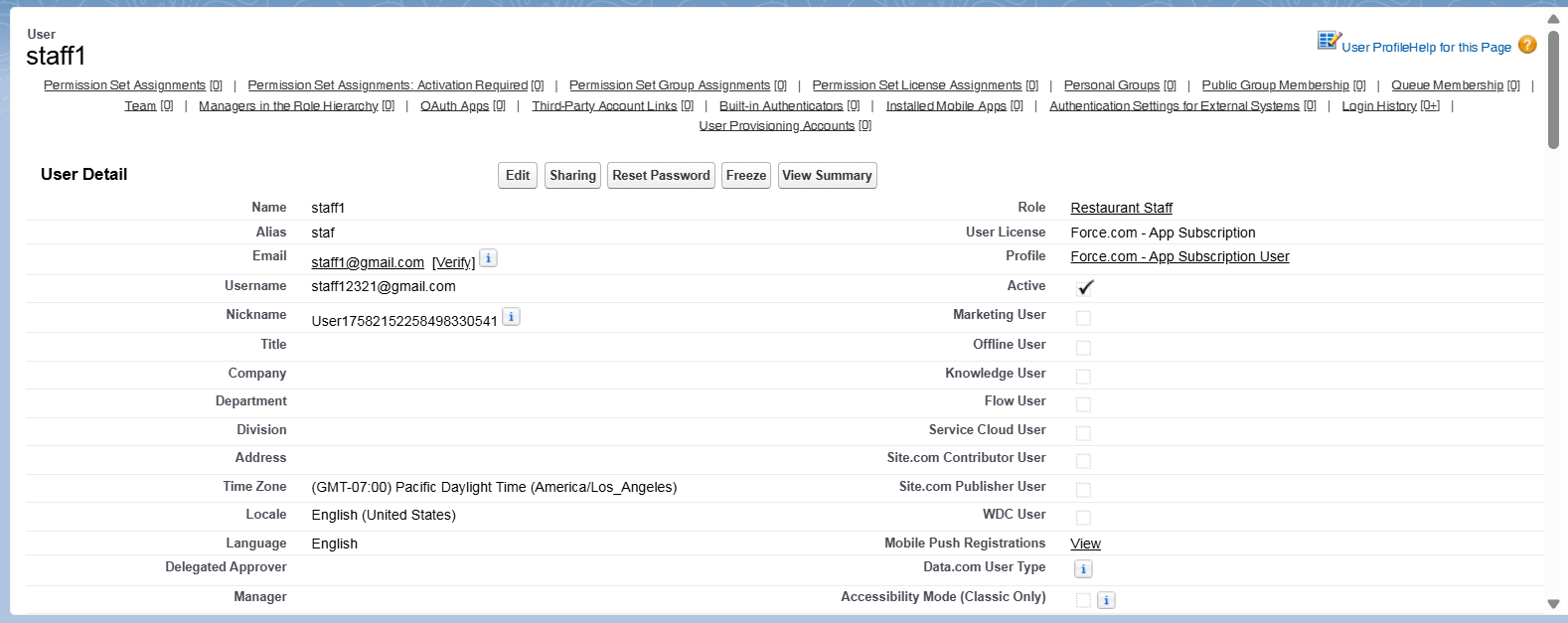
* **Profiles**: Custom profiles were created by cloning standard profiles to provide baseline access.
  + **Restaurant\_Staff\_Profile**: For daily order entry and customer interaction.



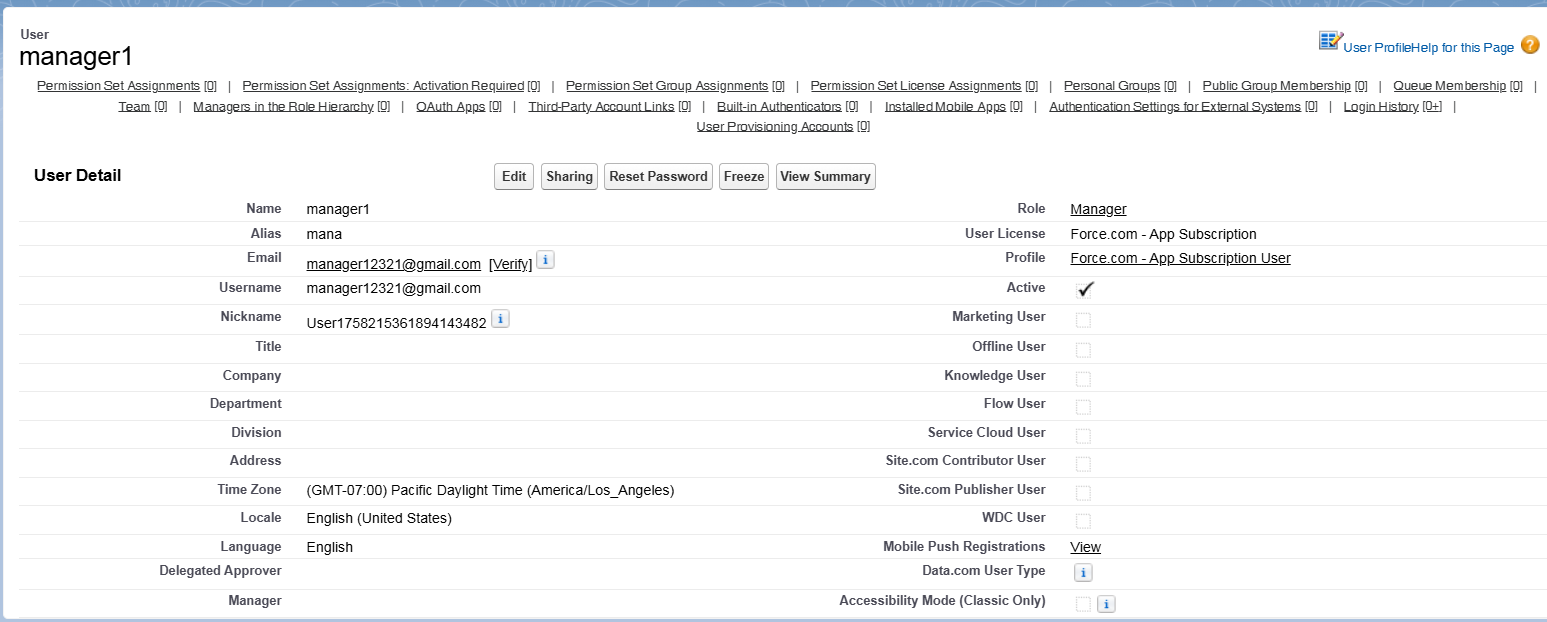
* + **Manager\_Profile**: For operational oversight, reporting, and manual adjustments.



* **Roles**: A role hierarchy was defined for data visibility and report roll-ups.
  + Manager, Restaurant Staff.
* **Permission Sets**: A permission set was created for special actions.
  + **Manual\_Points\_Adjust\_PS**: Grants access to modify loyalty points, assigned only to Managers.
* **Users**: Sample users were created and assigned the appropriate profiles and roles.
  + **staff1** → Restaurant\_Staff\_Profile, Role: Restaurant Staff



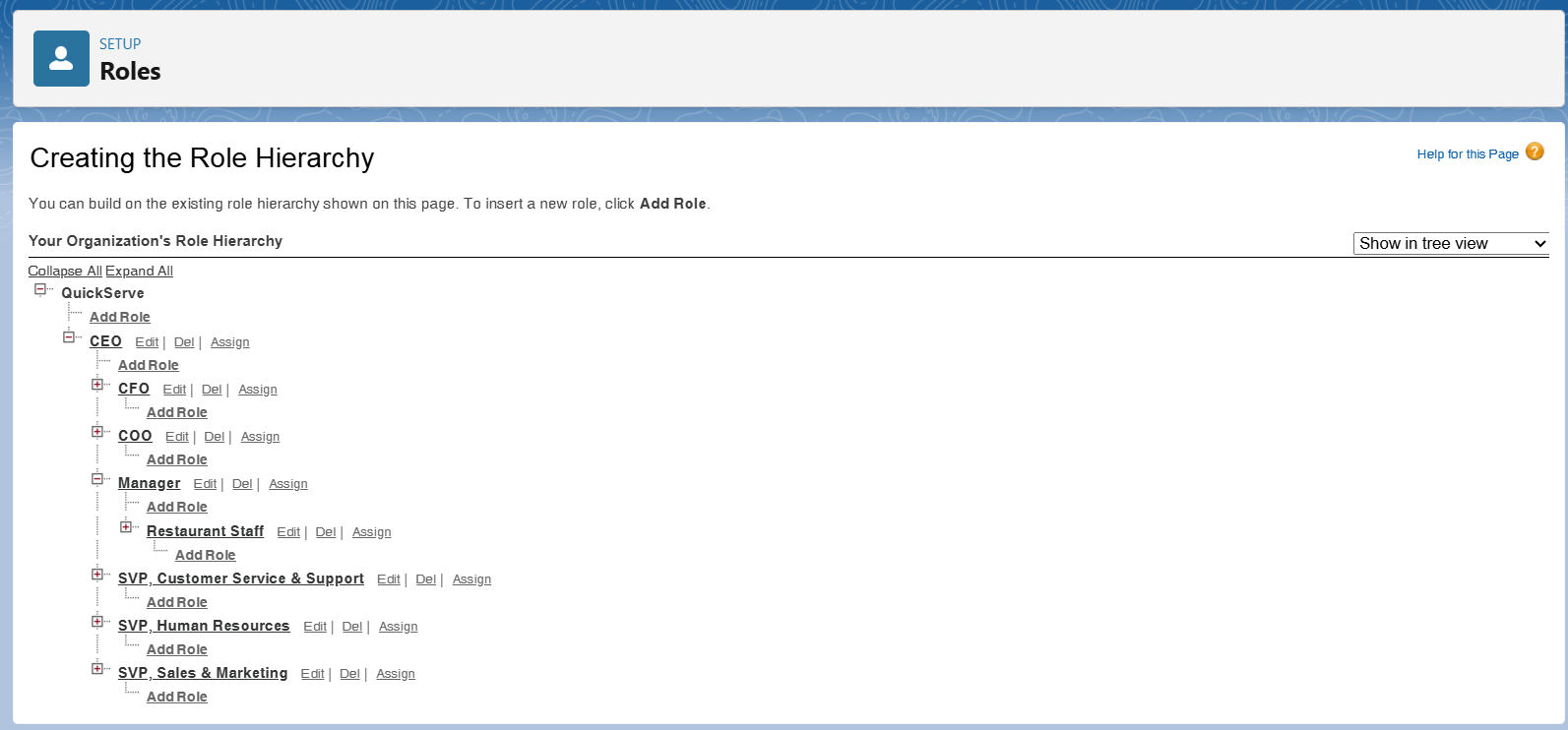
* + **manager1** → Manager\_Profile, Role: Manager



**♦️ Role Hierarchy Setup**

The hierarchy defines the data access and reporting structure within the organization.

* **Path**: Setup → Roles → Set Up Roles.
* **Top-Level Role**: CEO.
* **Manager Role**: Added as a child of the CEO.
* **Restaurant Staff Role**: Added as a child under the Manager.
* **Resulting Hierarchy**: CEO → Manager → Restaurant Staff



**♦️ OWD & Sharing Rules**

* **OWD (Org-Wide Default)**: This will be configured in Phase 3, as the settings need to be applied to the custom objects that will be created then.
* **Sharing Rule**: This will also be implemented in Phase 3 after the custom objects and OWD settings are in place.

**♦️ Dev Org Setup**

The development environment was prepared to support the project implementation.

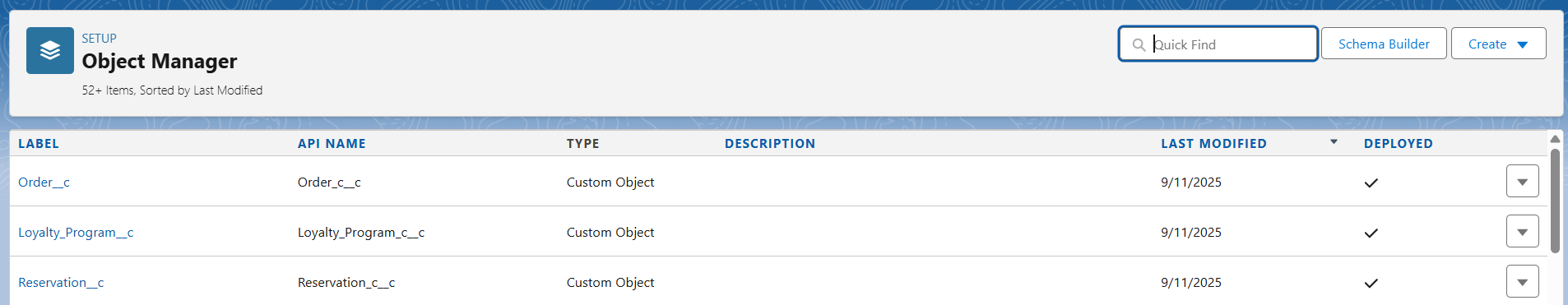
* A Salesforce **Developer Edition org** was set up to build the project.
* A **GitHub Repository** was created for version control of the source code.
* **VS Code and SFDX** were configured for developing the Apex backend and future Lightning Web Components.

**PHASE 3 - Data Modeling & Relationships**

**♦️ Standard & Custom Objects**

The data model utilizes both standard and custom objects to create a robust structure for the application.

* **Standard Objects:**
  + **Contact:** Represents the customer placing orders and making reservations.
* **Custom Objects:** Three new custom objects form the core of the application:
  + **Order\_\_c:** Tracks all customer transactions.
  + **Reservation\_\_c:** Manages table reservations.
  + **Loyalty\_Program\_\_c:** Holds customer loyalty point information.

****

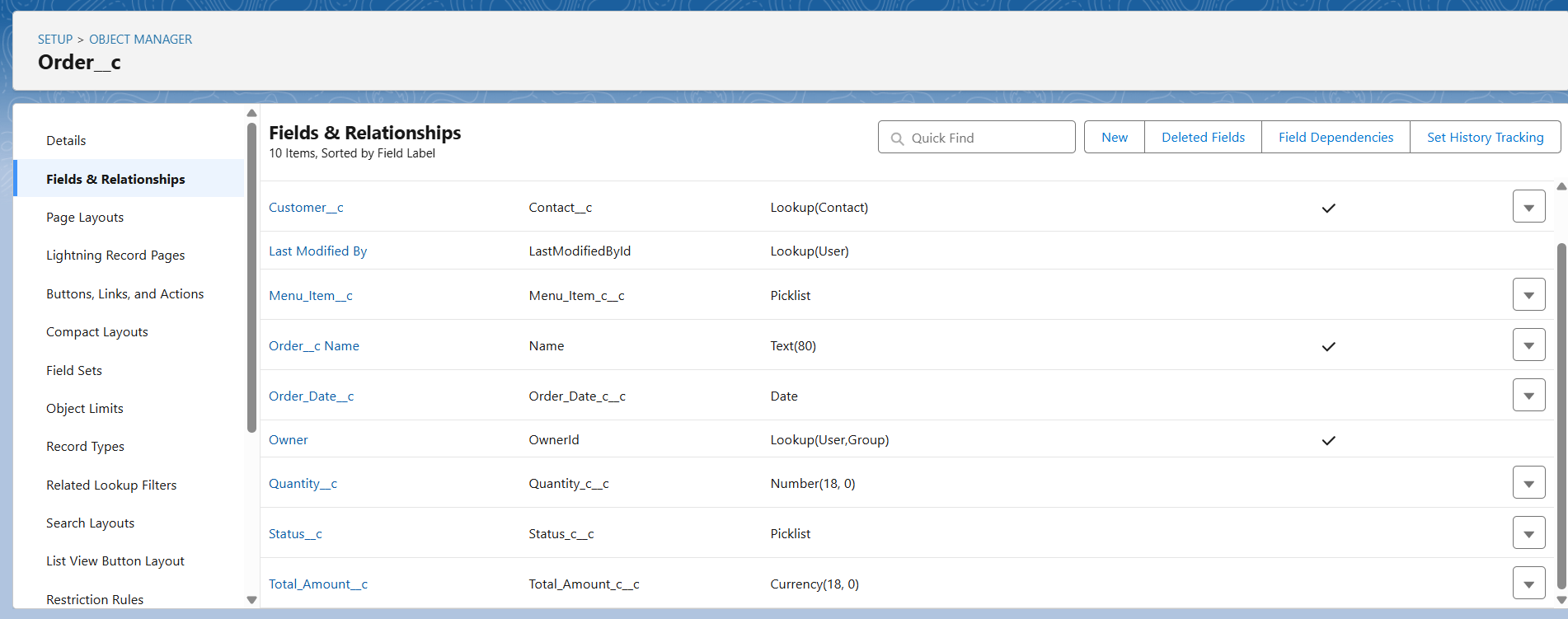
**♦️ Fields & Relationships**

Custom fields capture the required business data, and relationships link the objects together.

* **Relationships:**
  + **Customer\_\_c** (Lookup to Contact): All three custom objects **(Order\_\_c, Reservation\_\_c, Loyalty\_Program\_\_c)** have a lookup relationship to the Contact object. This creates a customer-centric model where all activities are tied back to a single contact record. Lookup was chosen over Master-Detail to allow these records to exist independently of a Contact if needed in the future.
* **Custom Fields:**

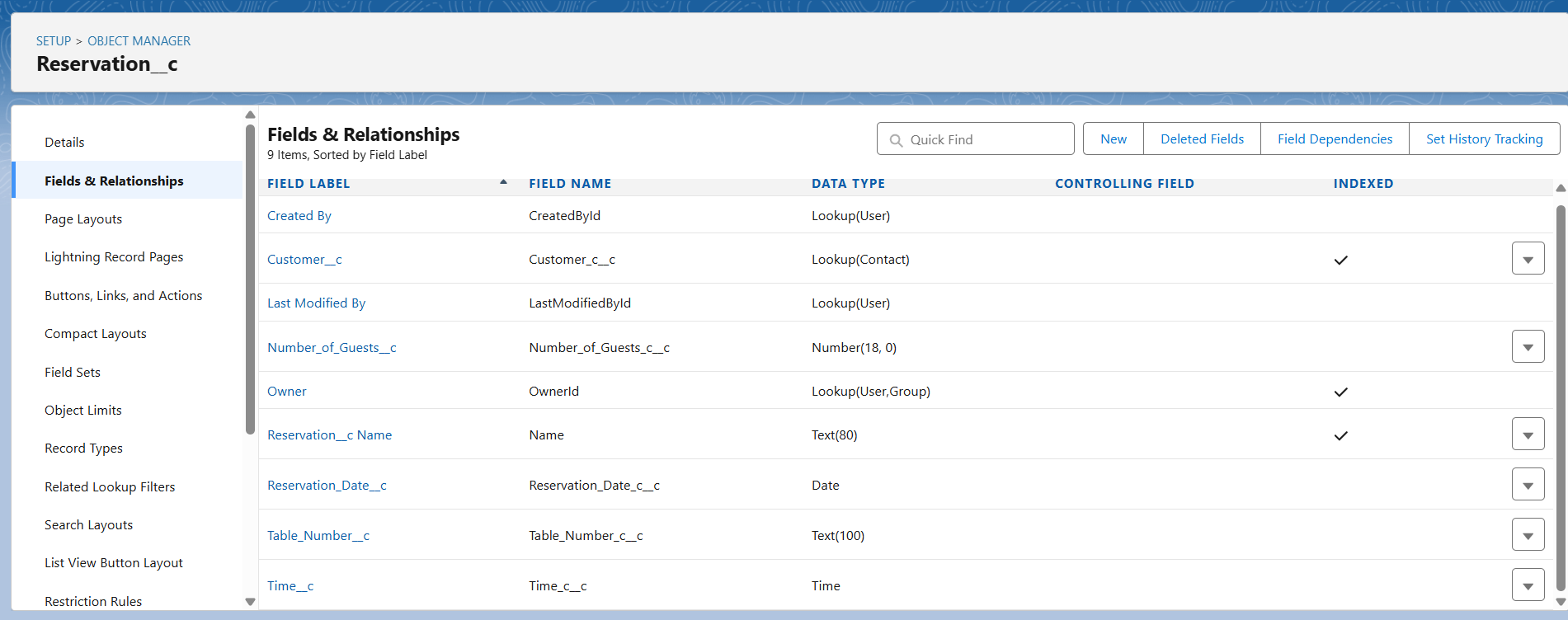
**On Order\_\_c:**

* Customer\_\_c (Lookup to Contact)
* Order\_Date\_\_c (Date)
* Menu\_Item\_\_c (Picklist)
* Quantity\_\_c (Number)
* Total\_Amount\_\_c (Currency)
* Status\_\_c (Picklist: 'Pending', 'Complete', 'Cancelled')

****

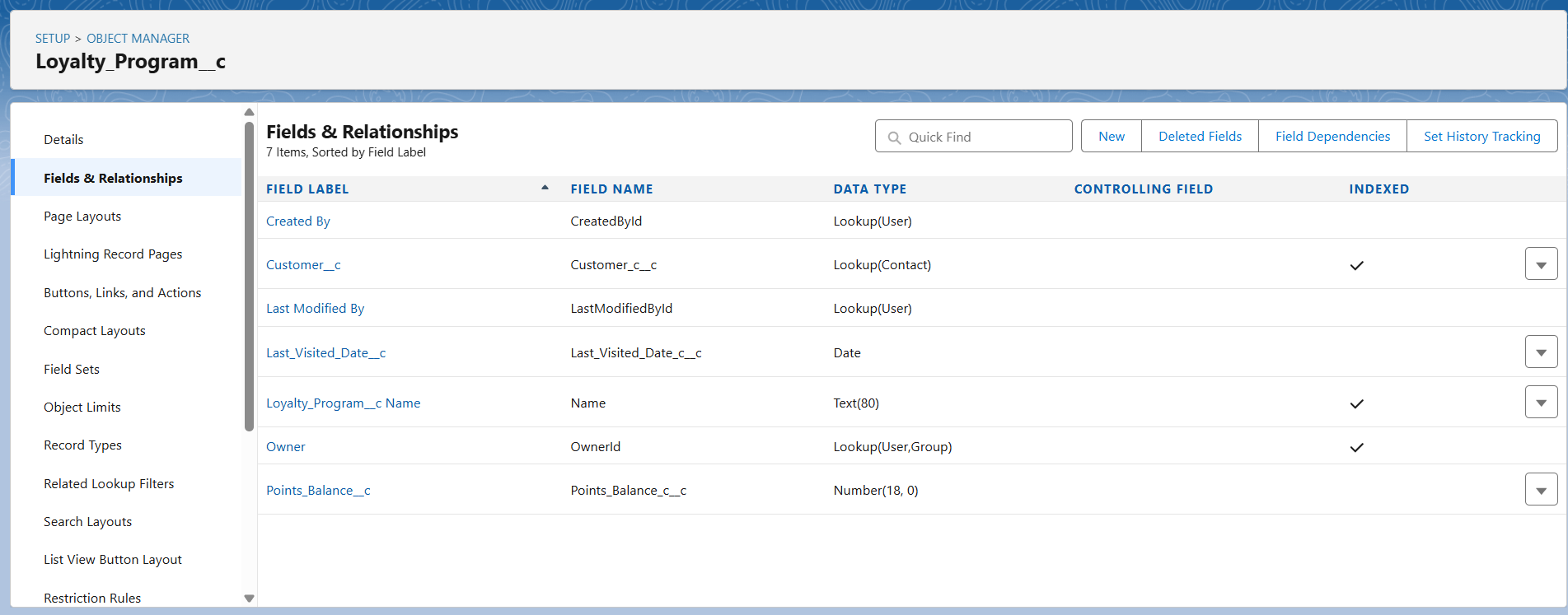
**On Reservation\_\_c:**

* Customer\_\_c (Lookup to Contact)
* Reservation\_Date\_\_c (Date)
* Time\_c (Time)
* Number\_of\_Guests\_\_c (Number)
* Table\_Number\_\_c (Text)

****

**On Loyalty\_Program\_\_c:**

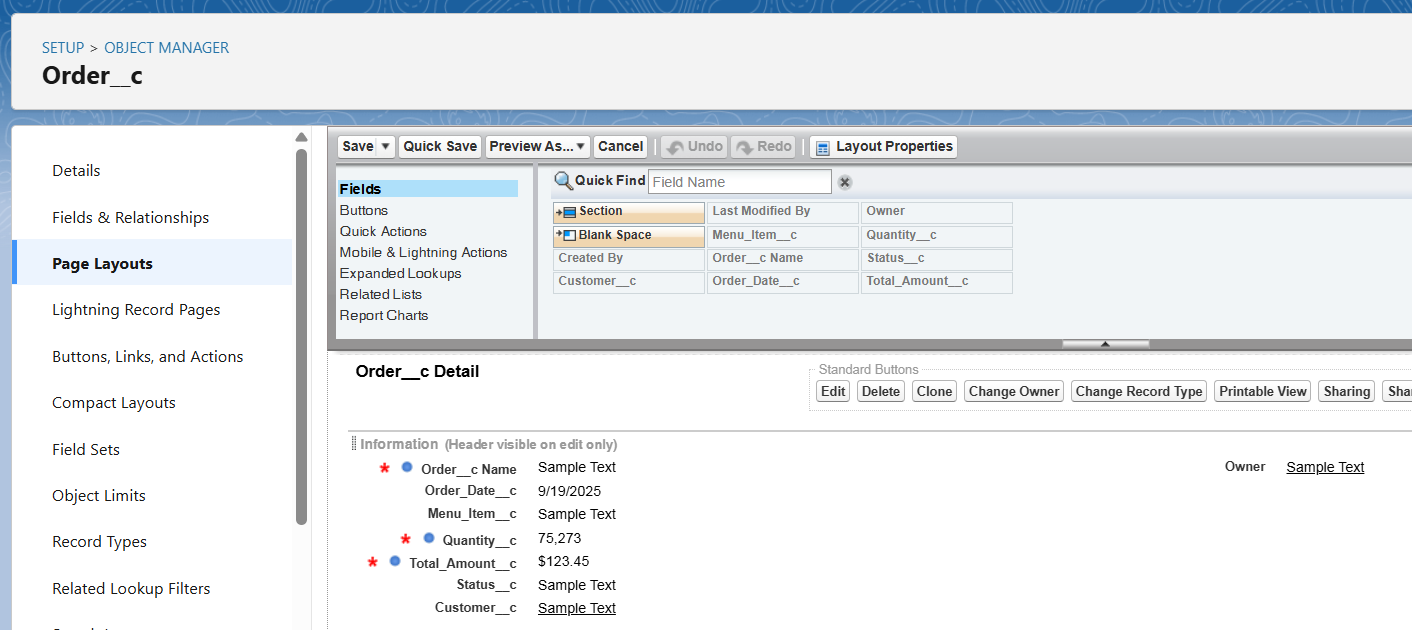
* Customer\_\_c (Lookup to Contact)
* Points\_Balance\_\_c (Number, Default Value: 0)
* Last\_Visited\_Date\_\_c (Date)

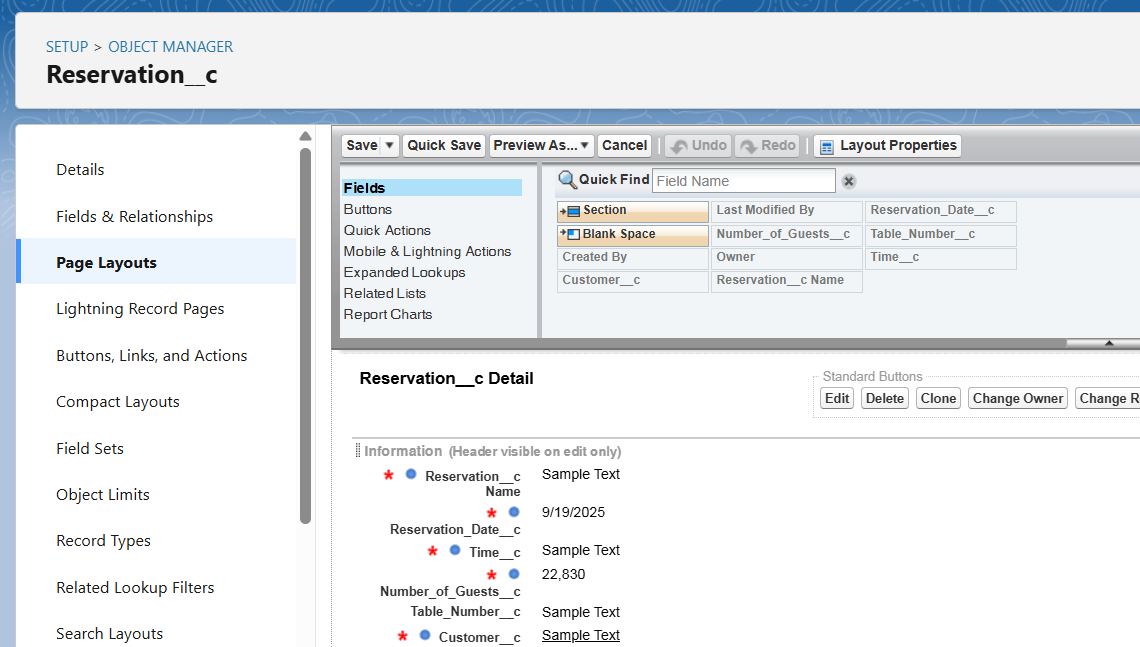


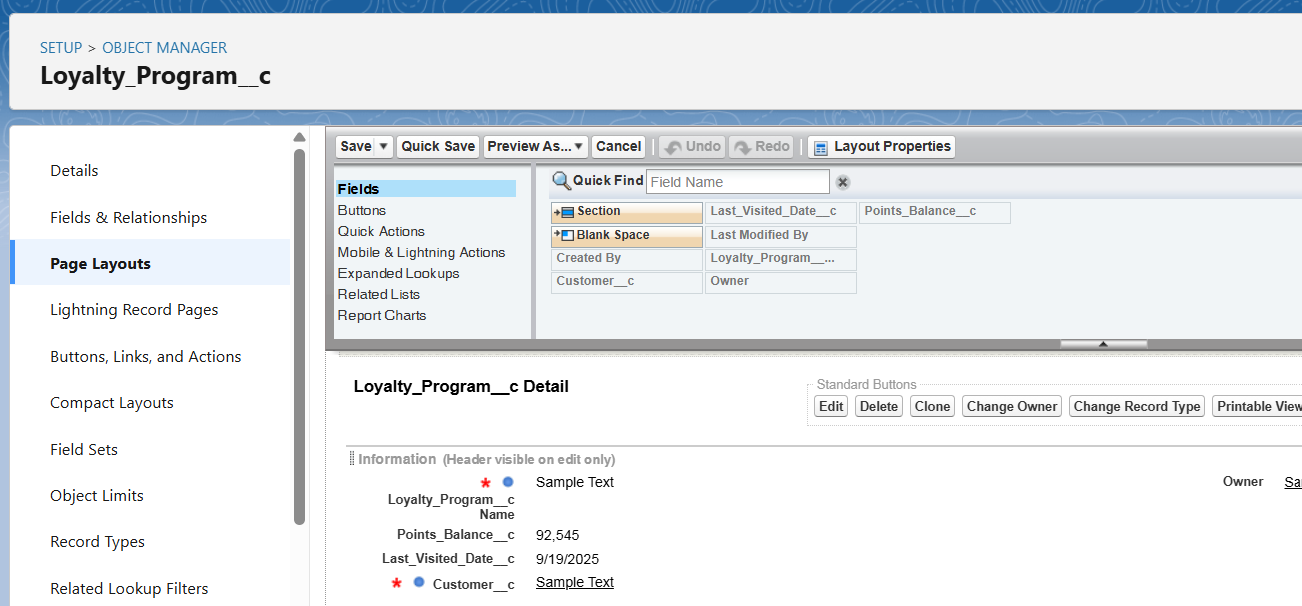
**♦️ User Interface & Layouts**

The on-screen display for records is controlled by various layout types.

* **Page Layouts:** A unique page layout for each custom object (Order, Reservation, Loyalty Program) will be created to organize fields and related lists for staff and managers.

****

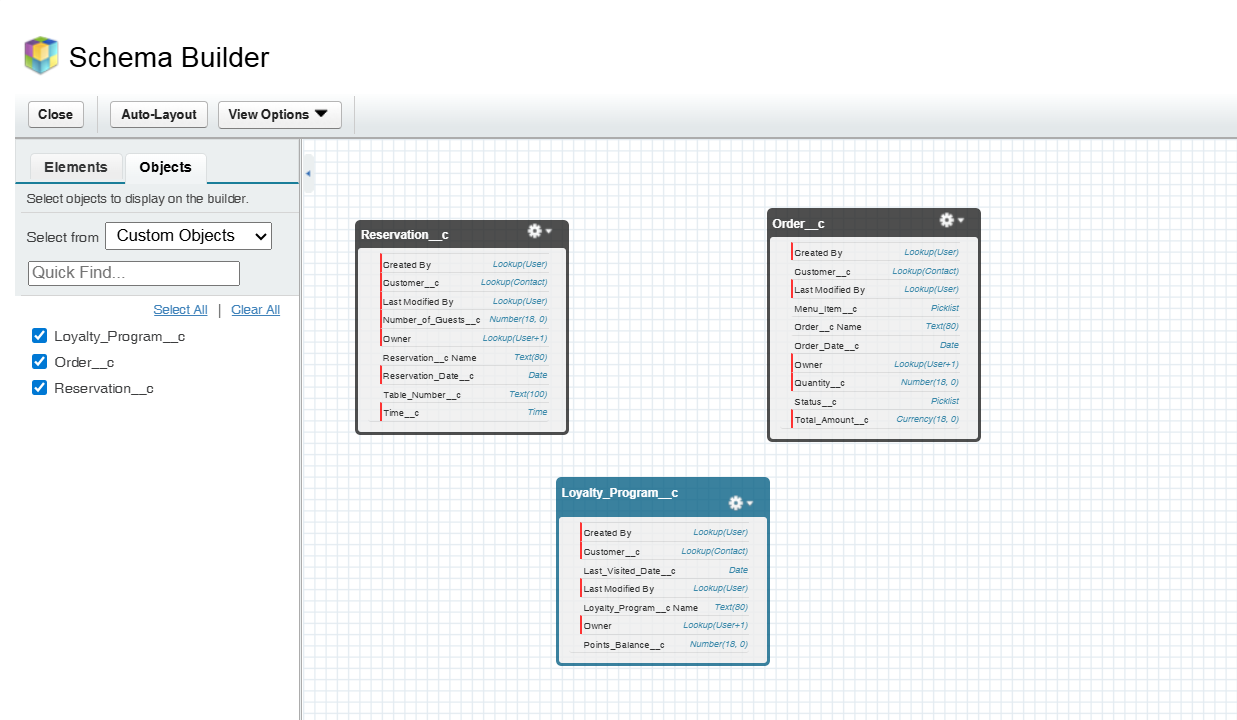
****

****

* **Compact Layouts:** A compact layout for each object will be configured to display key fields in the record highlights panel and in the Salesforce mobile app. For example, the **Order\_\_c** compact layout will show the **Status\_\_c** and **Total\_Amount\_\_c.**
* **Record Types:** Initially, a single record type will be used for each object. In a future enhancement, we could create record types on the **Order\_\_c** object like "Dine-In Order" and "Takeaway Order" to assign different page layouts or picklist values**.**

**♦️ Data Architecture**

* **Schema Builder:** The Schema Builder tool will be used to visualize the complete data model, showing the Contact object at the center with lookups from **Order\_\_c, Reservation\_\_c,** and **Loyalty\_Program\_\_c.**

****

* **Junction Objects:** A junction object is not required for the current design. However, if an **Order\_\_c** needed to contain multiple **Menu\_Item\_\_c** records, a junction object called Order\_Line\_Item\_\_c would be created with master-detail relationships to both **Order\_\_c** and a new **Menu\_Item\_\_c** custom object.

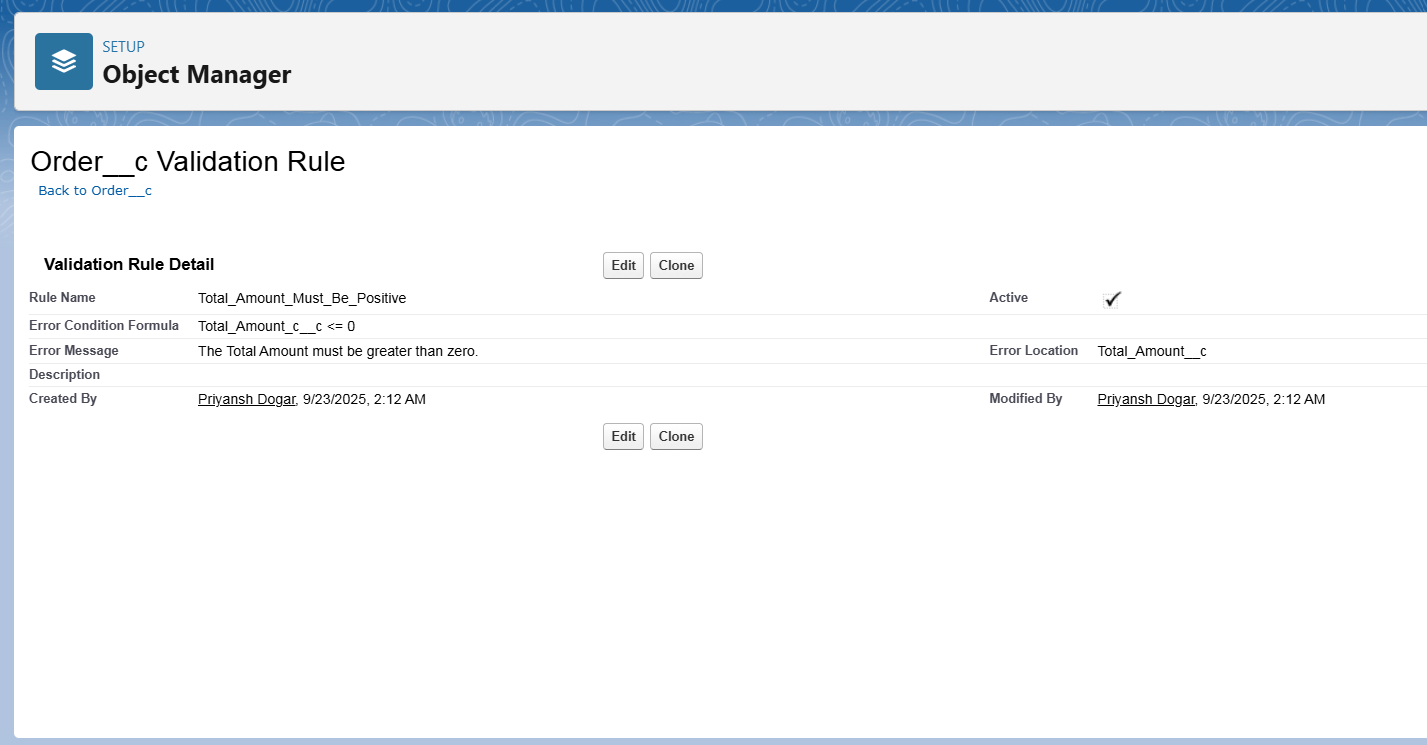
**PHASE 4 - Process Automation (Admin)**

The goal of this phase is to use Salesforce's declarative tools to automate the core business logic, enforce data quality, and handle notifications.

**♦️ Validation Rules**

Validation rules are created to ensure the data entered by users is accurate before it is saved.

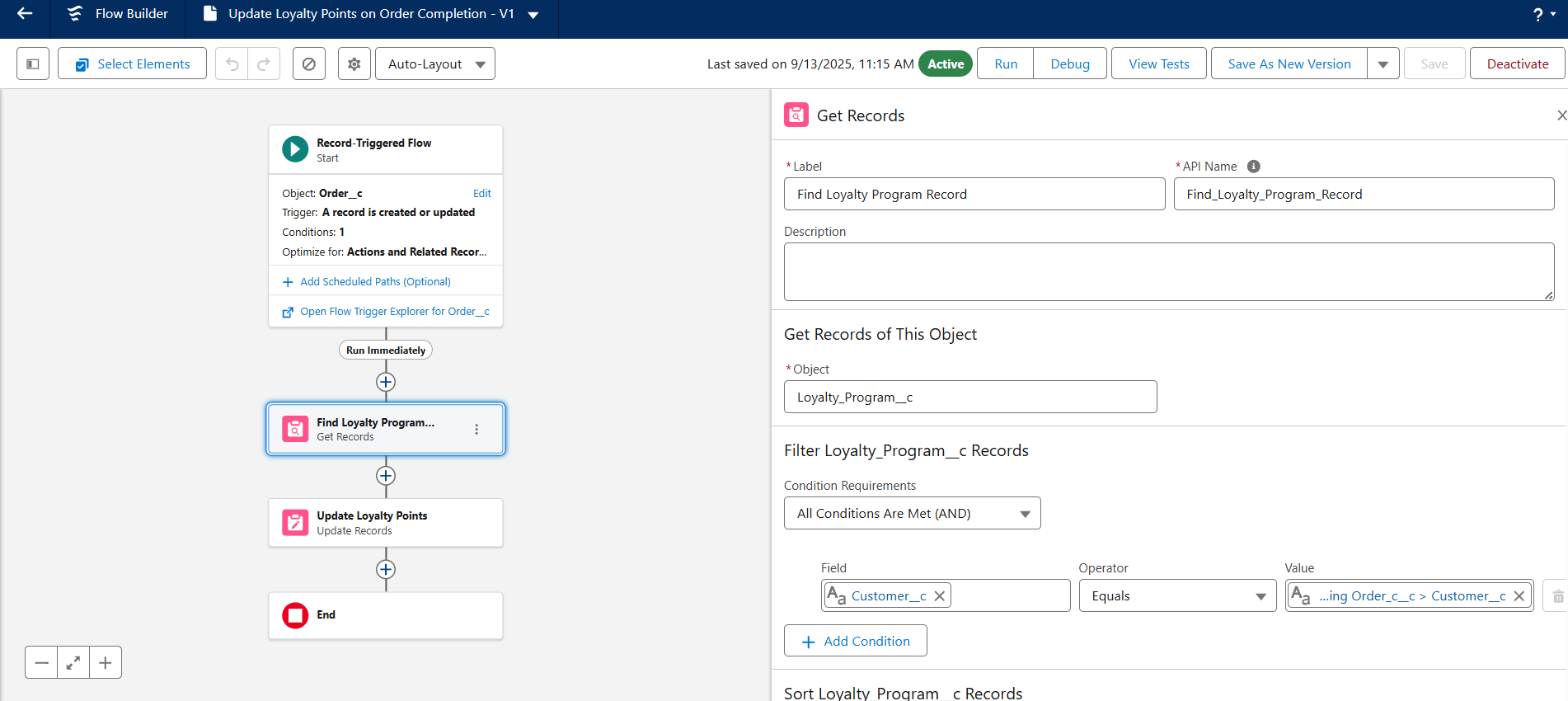
* **Object**: Order\_\_c
* **Rule**: To ensure the Total\_Amount\_\_c is a positive number.
* **Formula**: Total\_Amount\_\_c <= 0
* **Error Message**: "The Total Amount must be greater than zero."



**♦️ Flow Builder (Record-Triggered Flow)**

As the primary automation tool, Flow Builder was used to implement the core loyalty points logic. A record-triggered flow named "Update Loyalty Points on Order Completion" was built.

* **Objective**: To automatically find a customer's loyalty record and add points to it when their order is finalized.
* **Flow Type**: Record-Triggered Flow.
* **Trigger Configuration**: The flow is configured to run *after* an **Order\_\_c** record is **created or updated** and the **Status\_\_c** field **Equals 'Complete'**.
* **Flow Logic**:
  1. **Get Records**: The flow first finds the related Loyalty\_Program\_\_c record where its Customer\_\_c field matches the Customer\_\_c on the order that triggered the flow.



* 1. **Update Records (Field Updates)**: The flow then updates the Points\_Balance\_\_c field on the loyalty record. It uses a formula to add the new points (Total\_Amount\_\_c \* 0.1) to the existing balance.



* **Activation**: The flow is saved and activated to run automatically whenever the criteria are met.

**♦️ Approval Process**

An approval process automates how records are approved. **While not part of the initial build, a potential future use case has been identified**:

* **Objective**: Require a manager's approval for any Order\_\_c with a Total\_Amount\_\_c exceeding a certain value (e.g., 5,000 INR).
* **Process**: An order meeting the criteria would be automatically submitted to the user's manager for approval or rejection, updating the order's status accordingly.

**♦️ Workflow Rules & Process Builder**

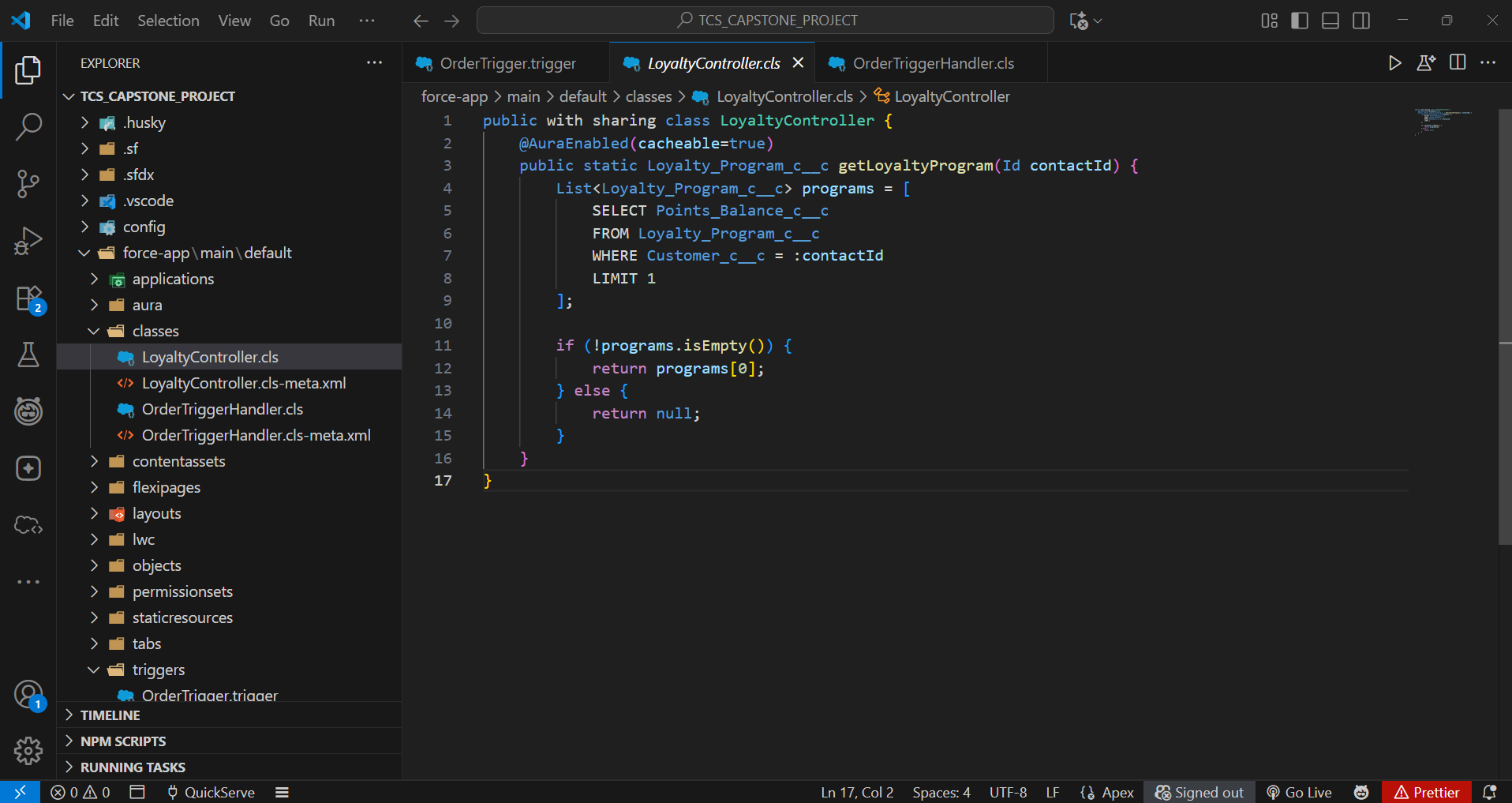
These are legacy automation tools in Salesforce. For this project, all new automation, including the loyalty points calculation, was built using **Flow Builder** to align with current Salesforce best practices.

**PHASE 5 - Apex Programming (Developer)**

This phase covers the custom backend logic built with Apex to handle the core requirements of the loyalty points calculation, which cannot be achieved with declarative tools alone.

**♦️ Classes & Objects**

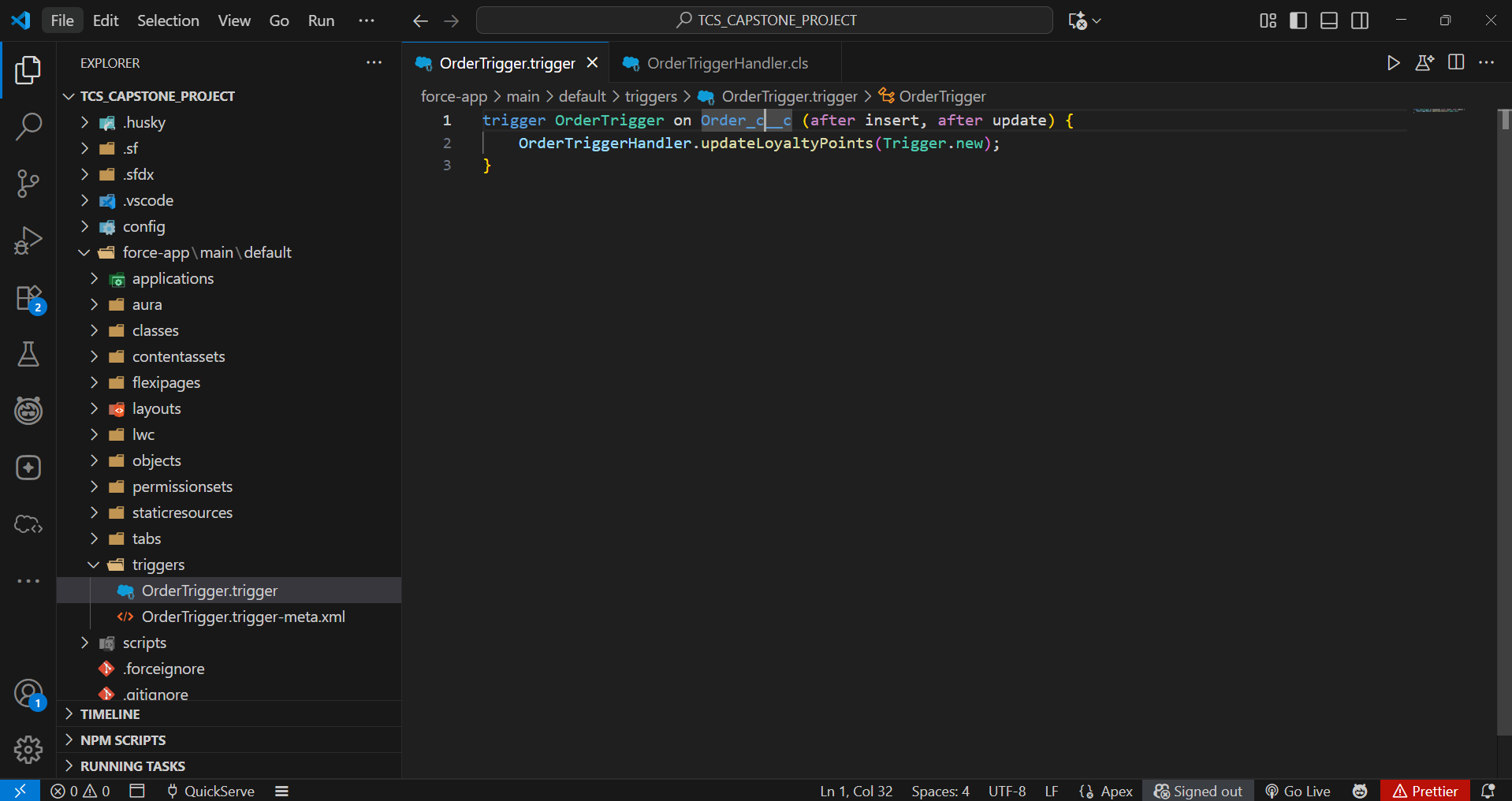
An Apex class LoyaltyController was created to serve data to the Lightning Web Component. This class interacts with our custom SObjects (Loyalty\_Program\_\_c, Order\_\_c) to query and return data.



**♦️ Apex Triggers**

The core automation is initiated by an Apex trigger.

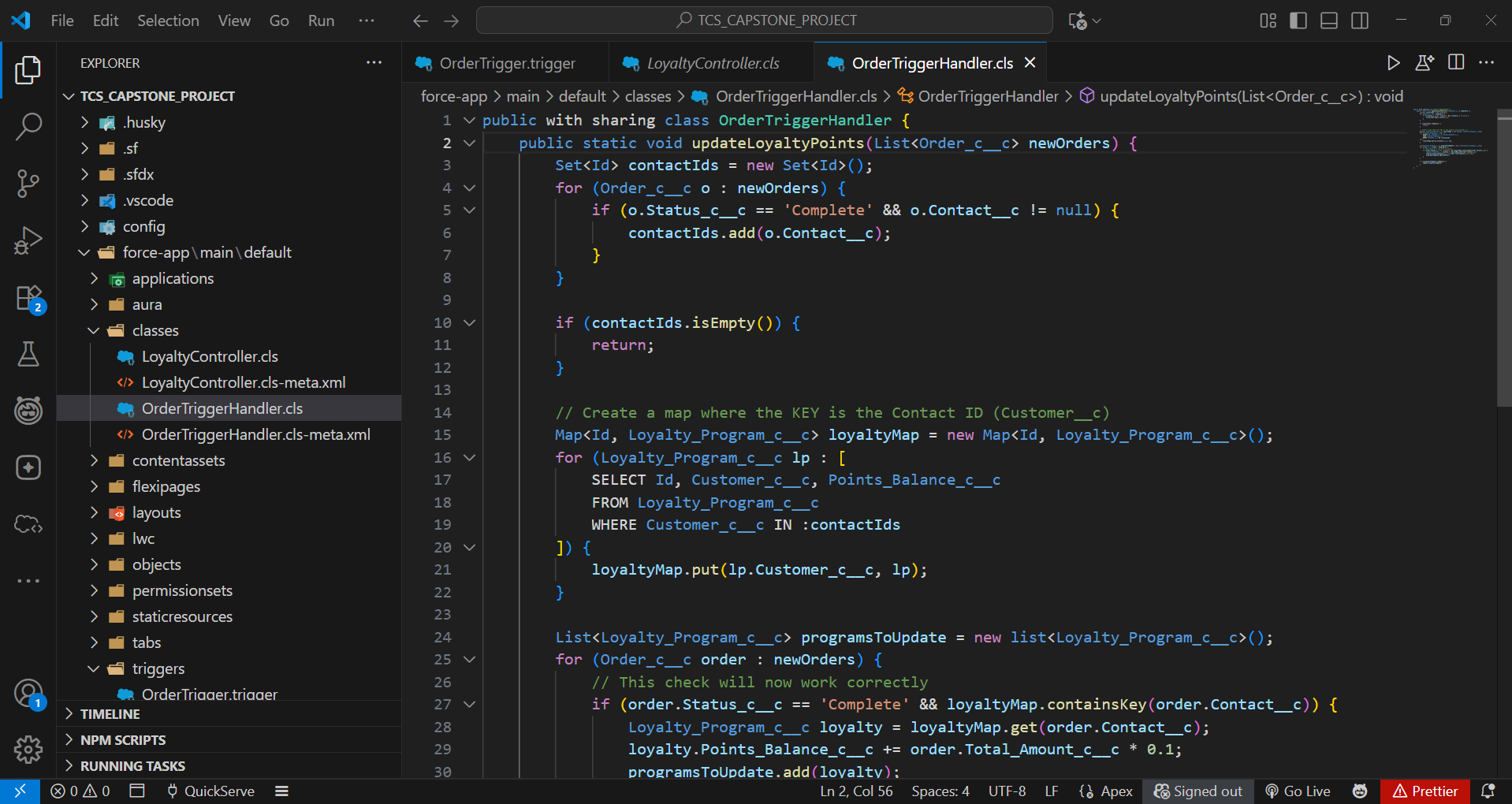
* **Trigger**: An OrderTrigger was created to run on the Order\_\_c object.
* **Events**: It is configured to fire after insert and after update to ensure the logic runs whenever an order is created or its status changes.



**♦️ Trigger Design Pattern**

A handler pattern was used to keep the trigger logic-less.

* The OrderTrigger does not contain any business logic itself.
* It calls a method in a separate handler class, OrderTriggerHandler, passing the trigger context variables (Trigger.new, Trigger.oldMap). This makes the code cleaner, more reusable, and easier to test.



**♦️ SOQL & Collections**

The Apex logic uses SOQL to query for records and Collections to manage the data efficiently.

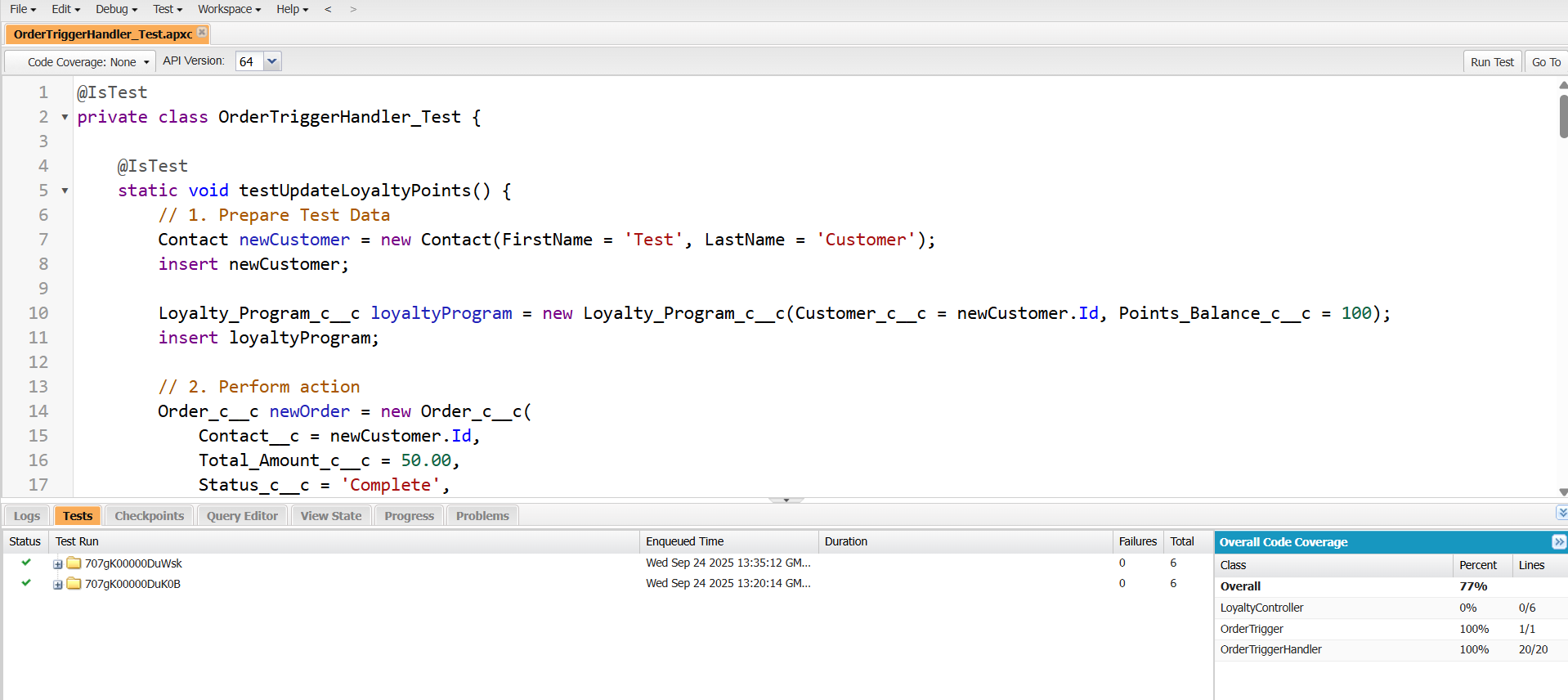
* **SOQL**: A SOQL query is used within the OrderTriggerHandler to select the Loyalty\_Program\_\_c record related to the customer on the Order\_\_c.
  + Example: SELECT Id, Points\_Balance\_\_c FROM Loyalty\_Program\_\_c WHERE Customer\_\_c = :contactId
* **Collections**:
  + A **Set** is used to collect unique Contact IDs from the orders to avoid duplicate queries.
  + A **Map** is used to hold the queried Loyalty\_Program\_\_c records, with the Contact ID as the key for easy retrieval.

**♦️ Exception Handling**

Try-Catch blocks are implemented within the Apex handler to gracefully manage potential errors, such as a query returning no records or a DML operation failing. This prevents the entire transaction from failing due to an unexpected issue.

**♦️ Test Classes**

A dedicated test class, OrderTriggerHandler\_Test, was created to validate the Apex logic.



* The test class creates all necessary data in isolation (Contacts, Orders, Loyalty Programs).
* It calls the handler method to simulate the trigger firing.
* **System.assertEquals()** is used to assert that the Points\_Balance\_\_c on the Loyalty\_Program\_\_c record was updated correctly after the order was set to 'Complete'.
* The class achieves over 75% code coverage, making the Apex trigger and handler ready for deployment.

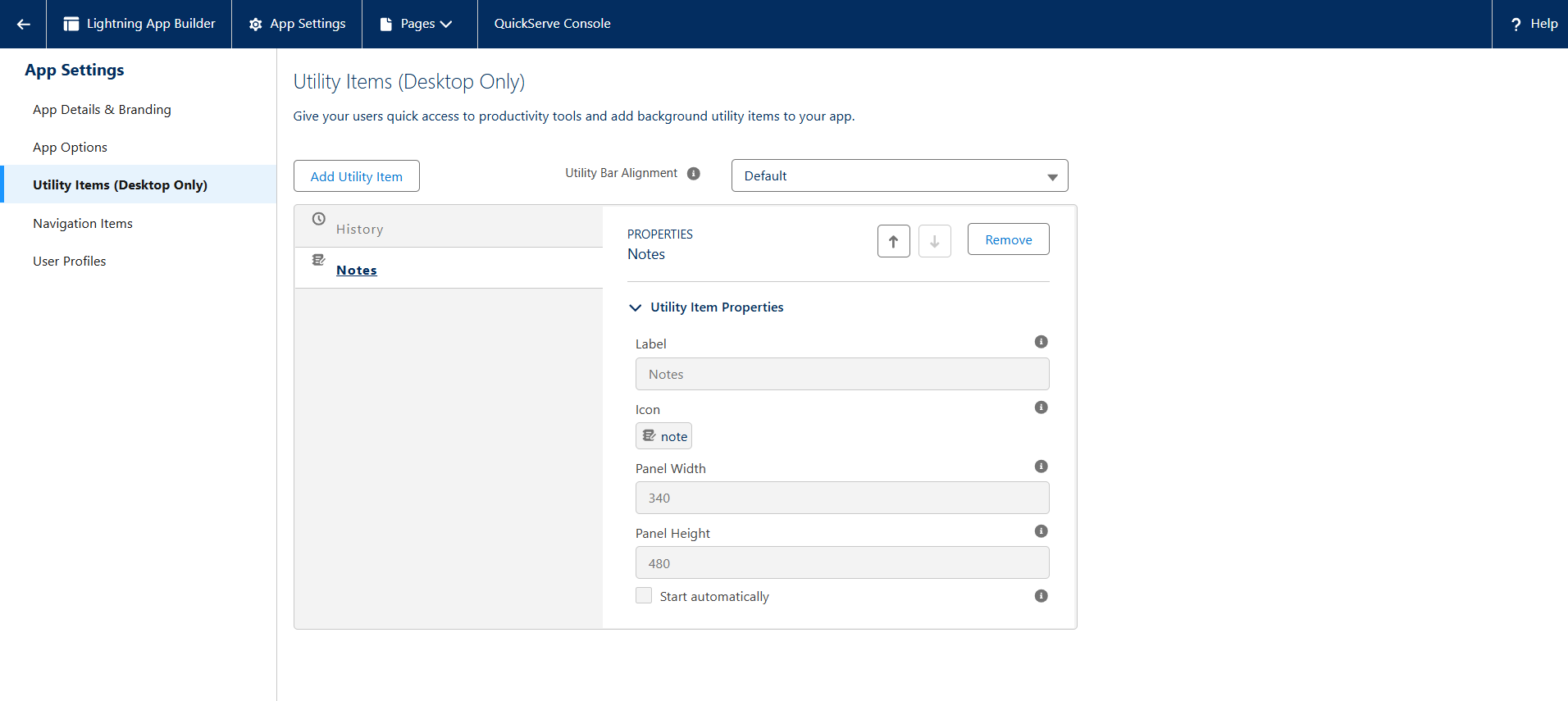
**PHASE 6 - User Interface Development**

This phase focuses on building a responsive and intuitive user interface using the Lightning App Builder and custom Lightning Web Components (LWC).

**♦️ Lightning App Builder**

The Lightning App Builder was the primary tool for creating the app and customizing pages declaratively.

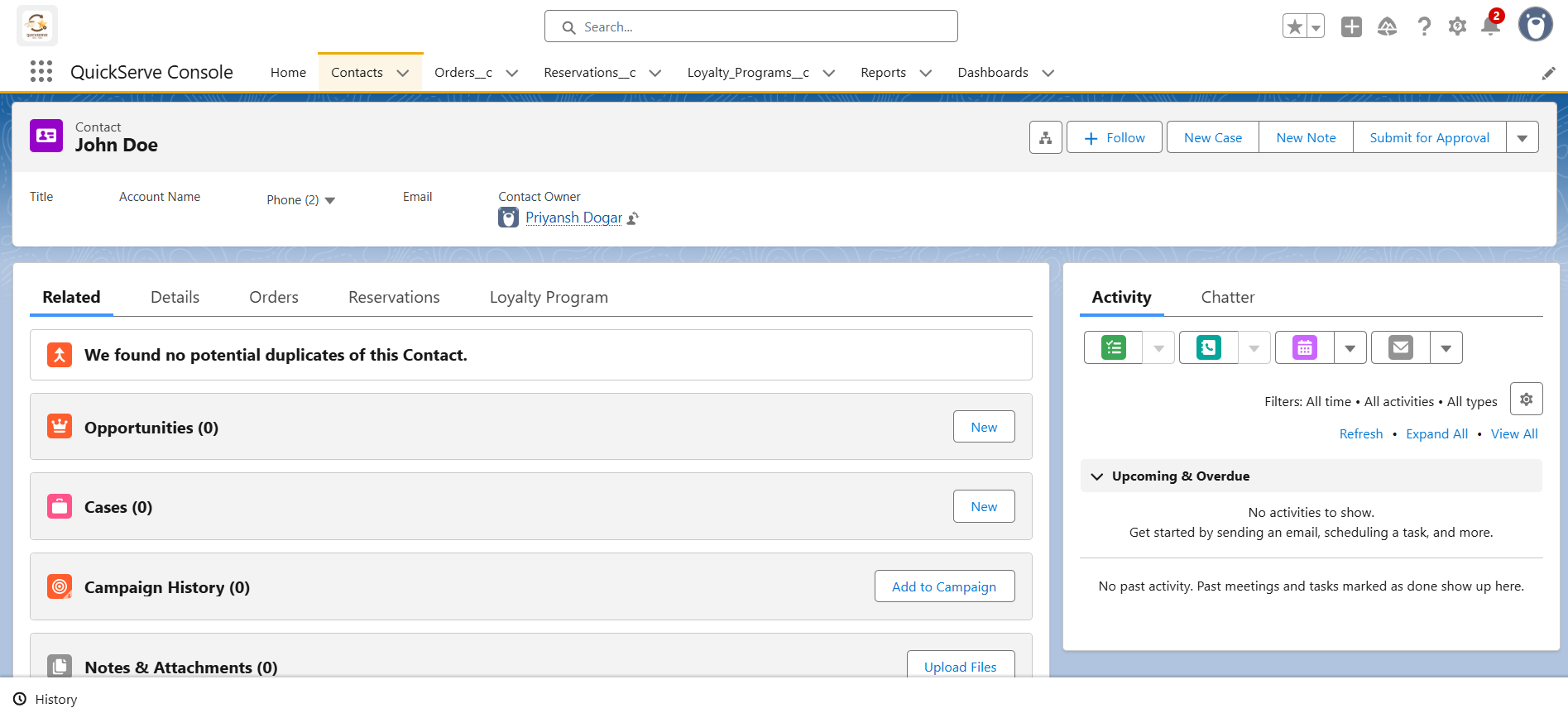
* **App Creation**: A dedicated Lightning App named "**QuickServe Console**" was built to provide a focused workspace for staff and managers.
* **Utility Bar**: Standard utilities like "History" and "Notes" were added to the app's utility bar for easy access to common tools.



**♦️ Record Pages & Tabs**

The standard Contact record page was enhanced to serve as the central hub for all customer-related activities.

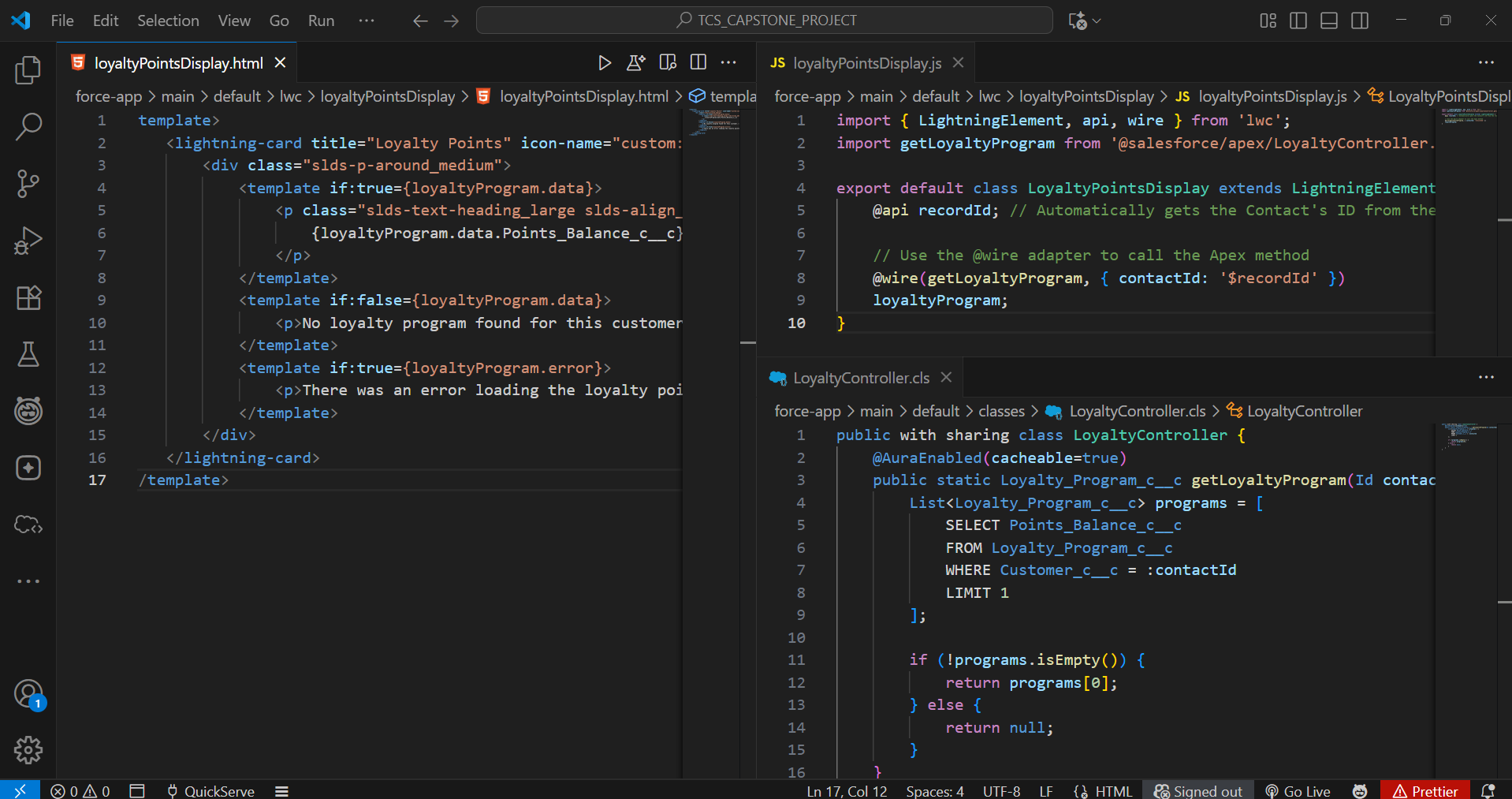
* **Custom Tabs**: The page was configured to include custom tabs for "Orders," "Reservations," and "Loyalty Program," each containing a related list to show the relevant records.



**♦️ Lightning Web Components (LWC)**

A custom Lightning Web Component (loyaltyPointsDisplay) was developed to provide a real-time view of a customer's loyalty status.

* **Apex with LWC**: The LWC communicates with a dedicated Apex class (LoyaltyController) to fetch data from the server. The Apex method is annotated with @AuraEnabled(cacheable=true) to make it securely accessible to the component.
* **Wire Adapters**: The component's JavaScript uses the @wire adapter to call the Apex method reactively. This automatically provides the data from the server and re-renders the component when the data changes.
* **Imperative Apex Calls**: While the display component uses a wire adapter, a future enhancement like a "Manual Points Adjustment" tool would use an imperative Apex call, as it needs to run in response to a user action (like a button click) and perform a DML operation.

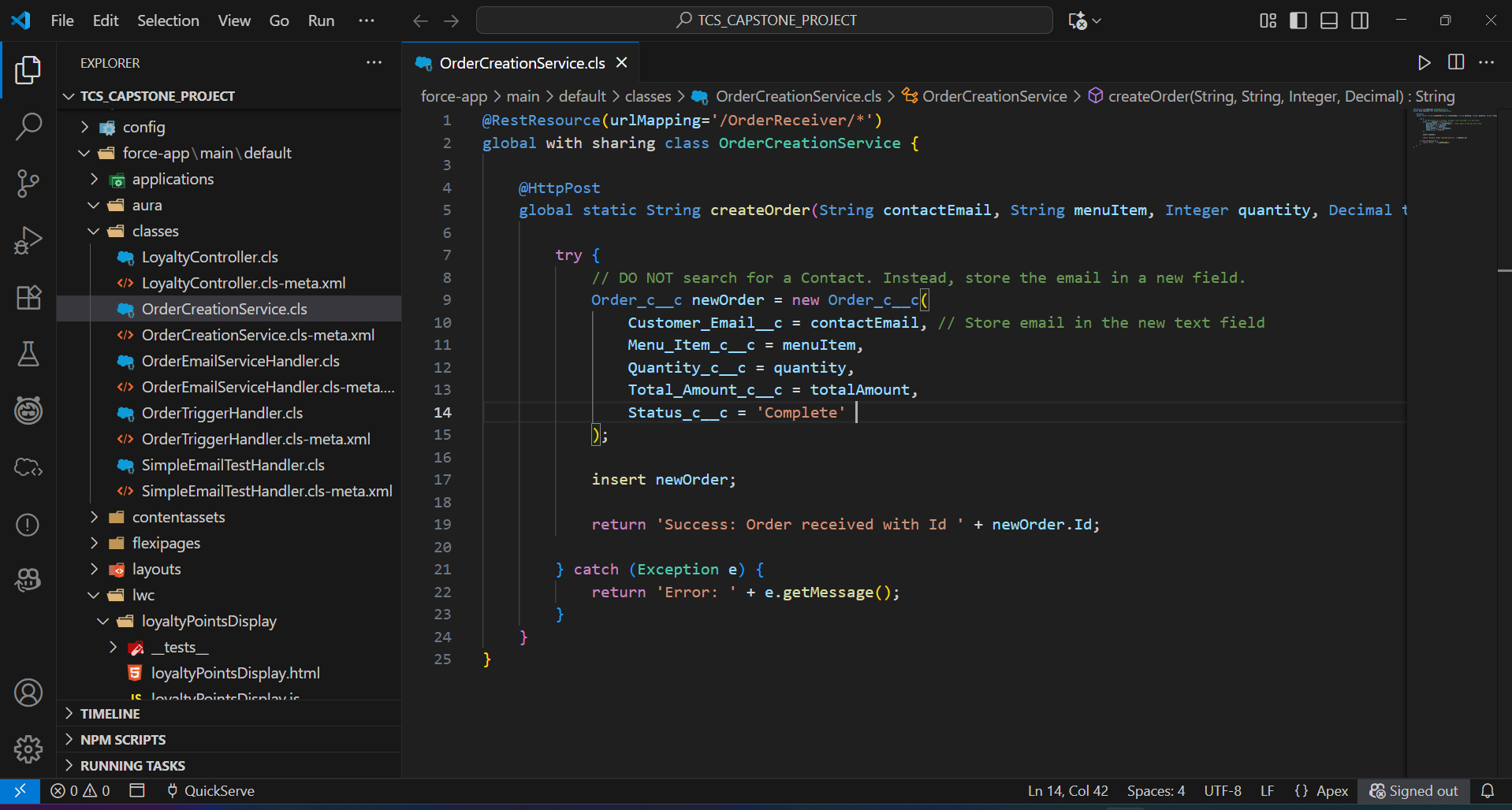


**PHASE 7 - Integration & External Access**

This phase focuses on enabling external systems, like a Point-of-Sale (POS) terminal, to create records in Salesforce. Due to org limitations preventing the use of a standard Connected App, an alternative integration method using a public web service was implemented.

**♦️ Web Services (REST)**

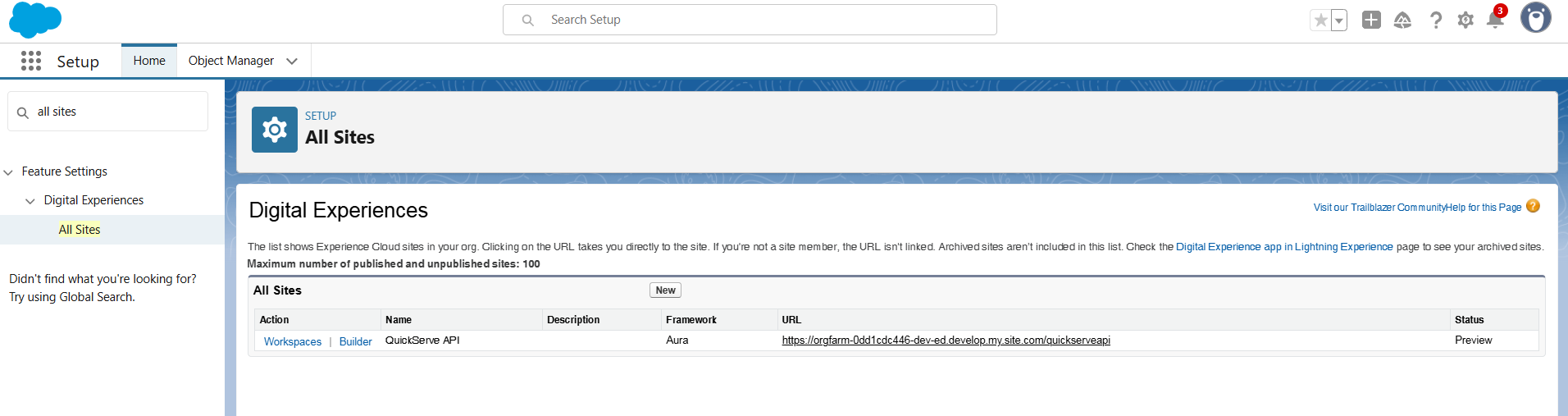
A custom Apex REST service was created to expose an endpoint that external systems can send data to. This allows for an "inbound" integration where a POS system can automatically create an Order\_\_c record in Salesforce.



* **Apex Class**: An @RestResource class named **OrderCreationService** was developed to handle incoming POST requests. The **@RestResource** annotation exposes the class as a web service, and the **@HttpPost** annotation specifies that a method will run when it receives an HTTP POST request.
* **Functionality**: The service is designed to receive a JSON payload containing order details (contact email, menu item, quantity, total amount). It then deserializes this data, creates a new Order\_\_c record, and returns a success or error message.

**♦️ Public Endpoint (Digital Experience)**

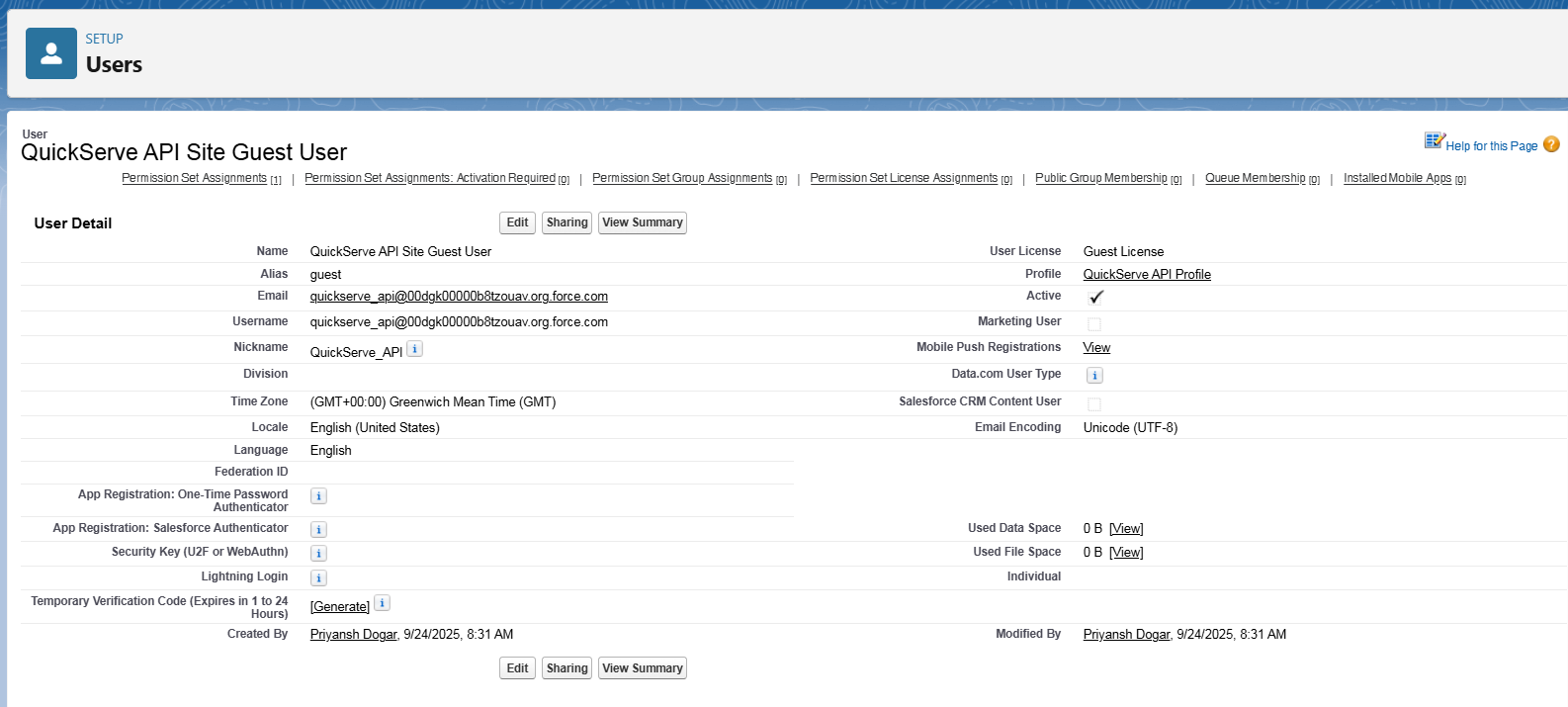
Instead of a private API secured by OAuth, a public endpoint was created using a Salesforce Site (Digital Experience). This provides a publicly accessible URL for the web service.



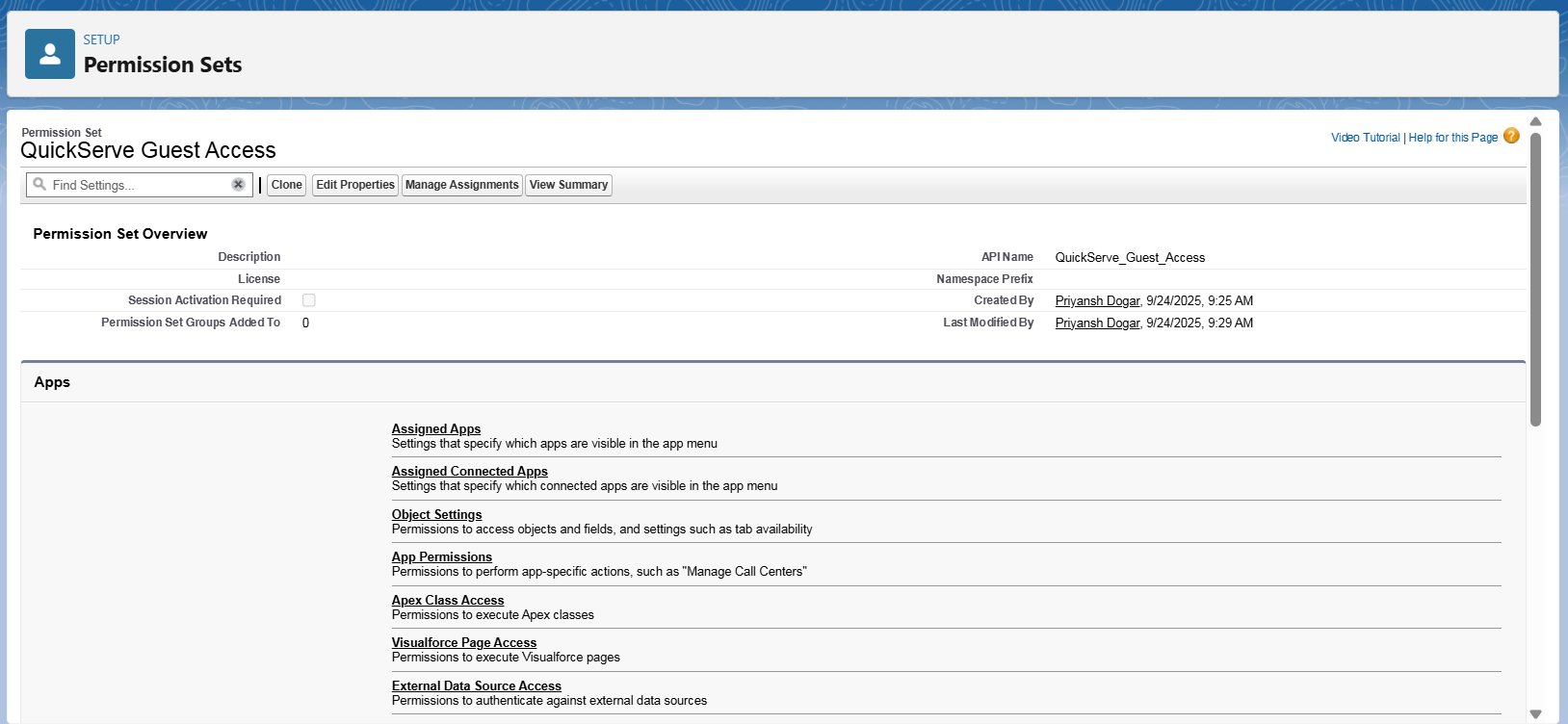
* **Site Setup**: A public Digital Experience site named "QuickServe API" was created to host the web service. This gives the endpoint a stable, public-facing URL.
* **URL Mapping**: The OrderCreationService was exposed via a custom URL mapping (/OrderReceiver/\*) on this public site. This means the final endpoint becomes <https://orgfarm-0dd1cdc446-dev-ed.develop.my.site.com/quickserveapi/services/apexrest/OrderReceiver/>

**♦️ Security & Permissions**

Since a standard Connected App could not be used, security was configured using the Guest User Profile and a dedicated Permission Set. This aligns with Salesforce's enhanced security model for public sites.



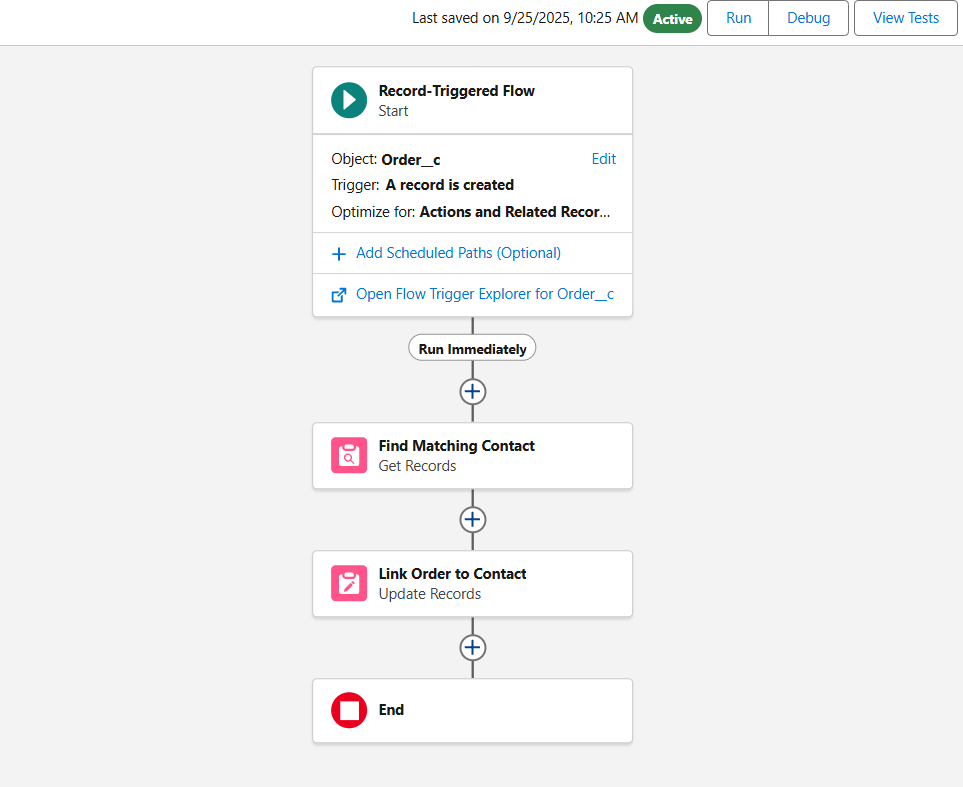
* **Guest User Profile**: The site has a specific "QuickServe API Profile" for unauthenticated users. This profile is intentionally kept with minimal permissions.



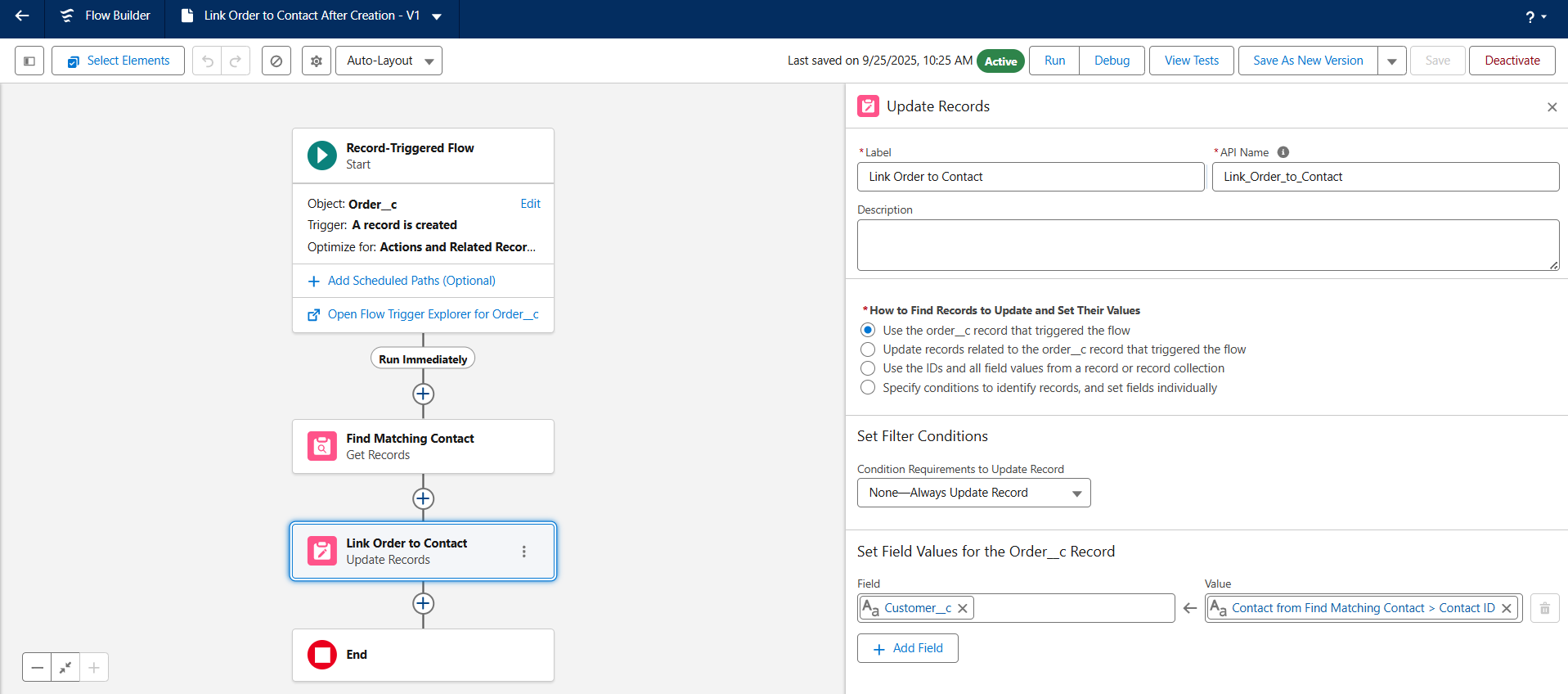
* **Permission Set**: A Permission Set named "QuickServe Guest Access" was created to securely grant the Guest User Profile explicit **Create** access on the Order\_\_c object and **Read** access on the Contact object, along with the necessary field-level security. This is the modern, secure way to grant permissions to guest users, rather than assigning them directly to the profile.

**♦️ Record Linking Automation**

A key challenge is that the secure Guest User cannot query for existing Contact records. A two-step automation process was built to bypass this limitation.



1. The **OrderCreationService** runs in the guest user's context. It creates an Order record but only stores the customer's email in a temporary text field, without linking it to a Contact.
2. An after-save **Record-Triggered Flow** then runs in **System Context**. Because it runs in System Context, it has permission to see all records. It takes the email from the new Order record, queries for the matching Contact, and then populates the Customer\_\_c lookup field, creating the final link.



**♦️ API Limits**

The integration was designed with Salesforce's strict governor and API limits in mind.

* The current Apex service creates one record at a time, which is safe for low-volume transactions. For any future high-volume needs, the service would be enhanced to handle a list of orders in a single transaction (bulkification) to avoid hitting execution limits.

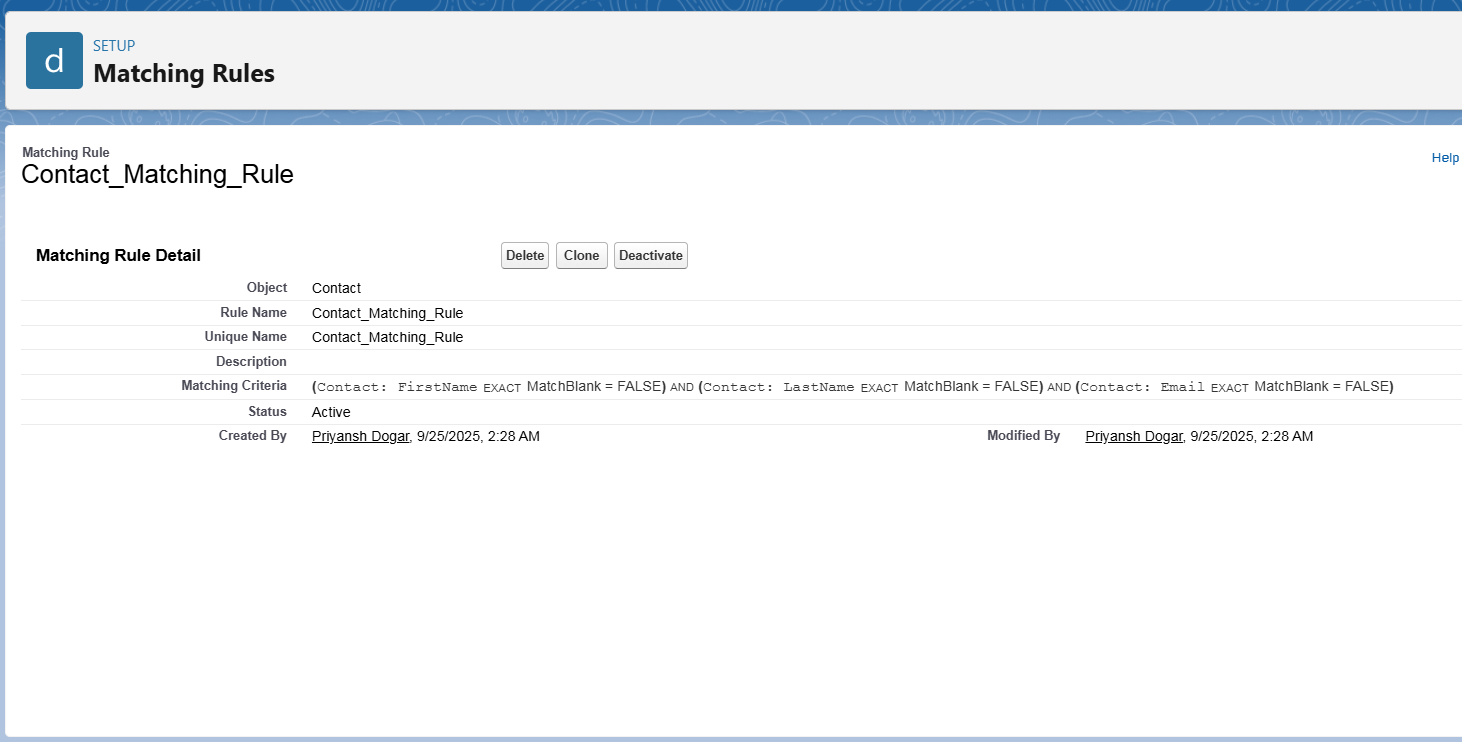
**PHASE 8 - Data Management & Deployment**

This phase focuses on the critical processes of ensuring long-term data quality, populating the org with initial data sets, and defining the strategy for migrating the application's components from the development environment.

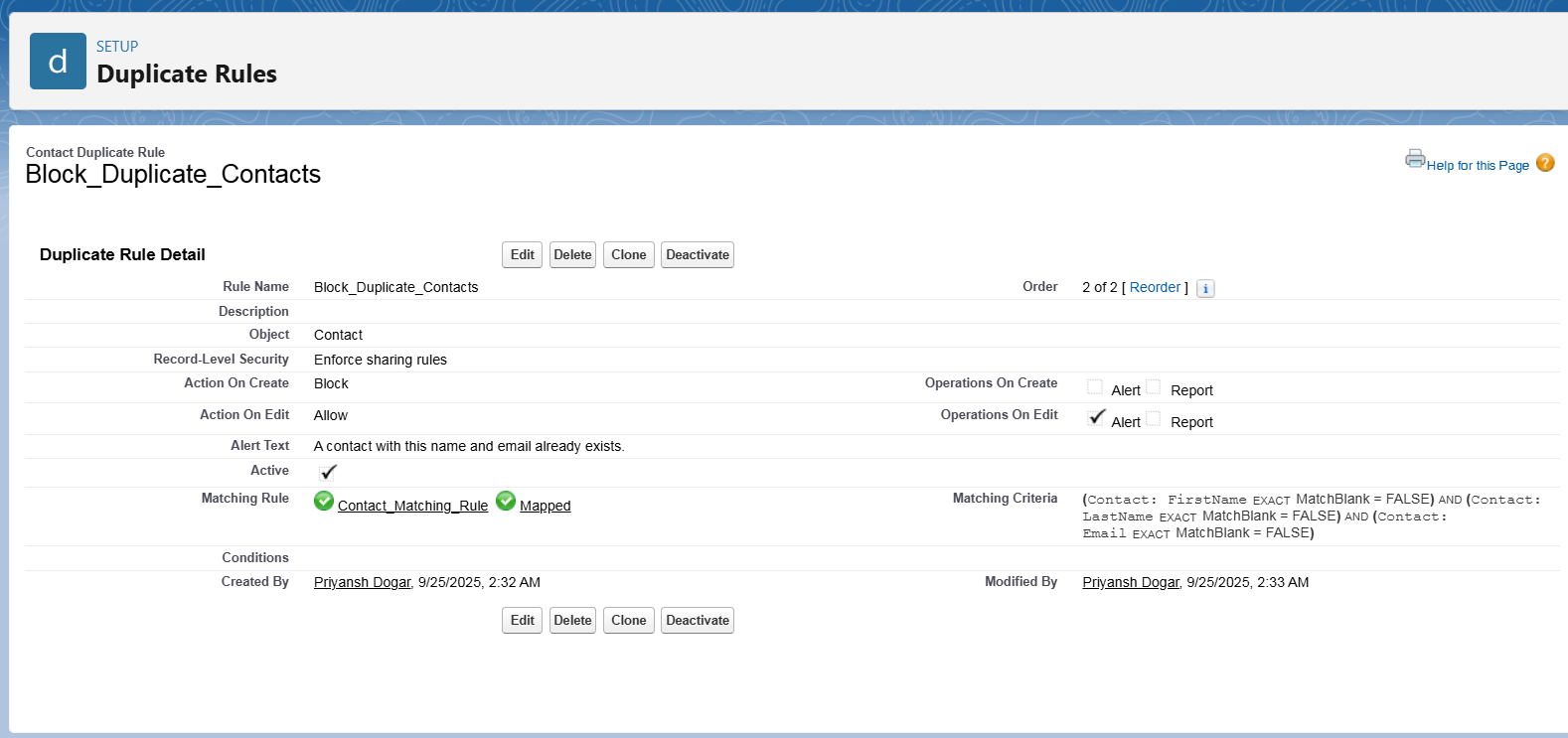
**♦️ Duplicate Rules**

To maintain a clean and reliable customer database, a robust duplicate management strategy was implemented for the Contact object. This prevents redundant records, ensuring a single view of the customer.

* **Steps Taken:**
  1. A **Matching Rule** was created to define a duplicate as an exact match on a Contact's First Name, Last Name, and Email.



* 1. A **Duplicate Rule** was then created to enforce the matching rule.

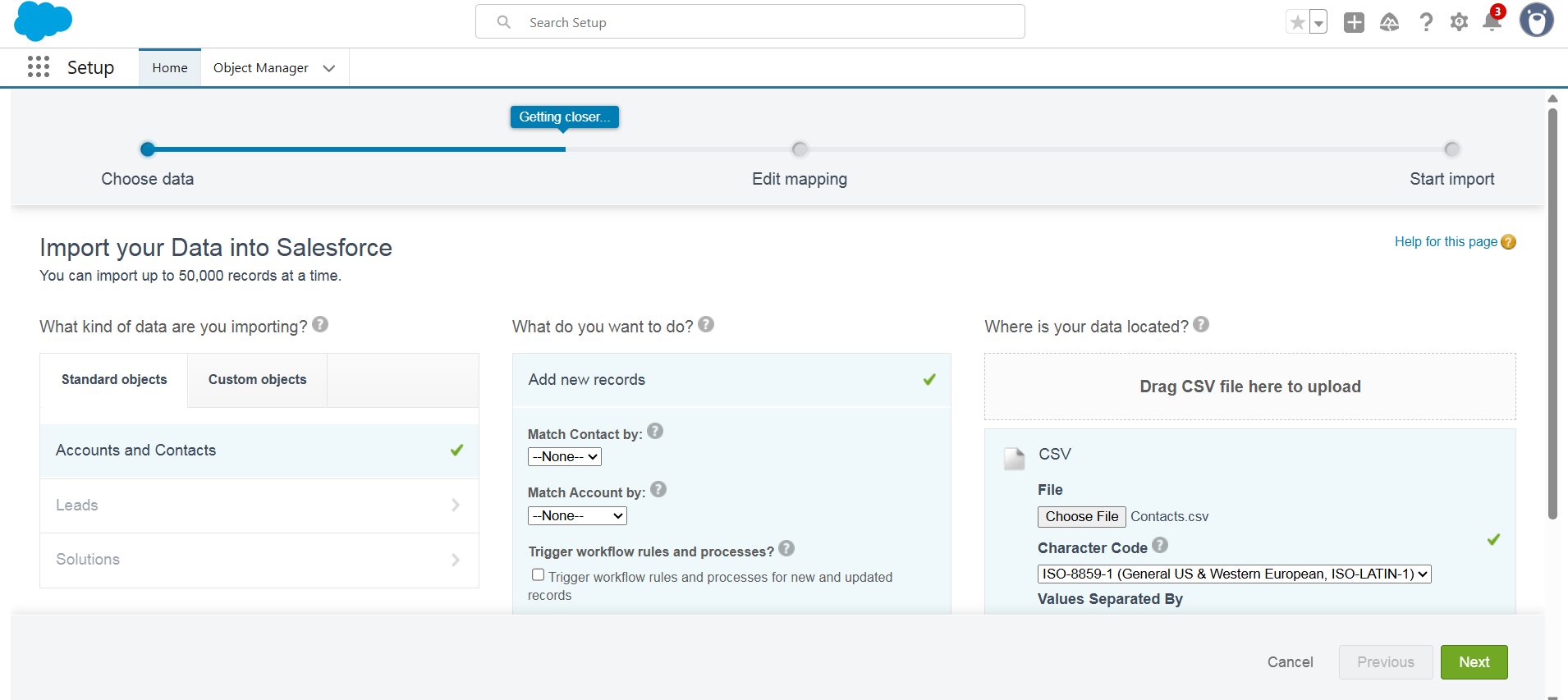


* 1. The rule was configured to **Block** users from creating new duplicates and **Allow** (but alert) on edits, balancing data integrity with user experience.

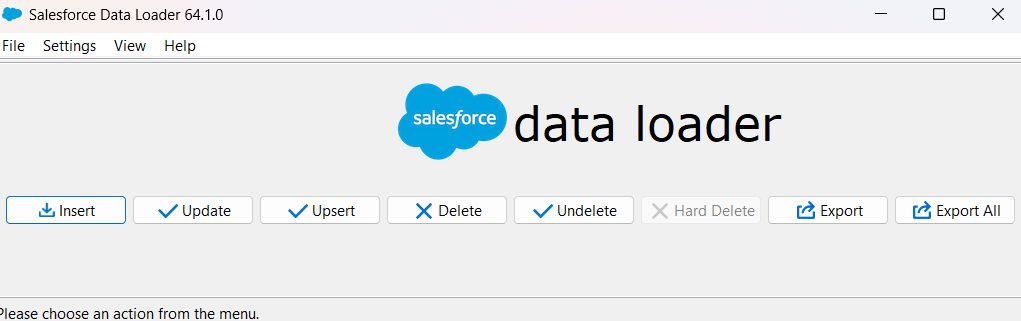
**♦️ Data Import Wizard & Data Loader**

Both of Salesforce's primary data import tools were used strategically to populate the org with sample data based on the specific requirements of each object.

* **Steps Taken:**
  1. A Contacts.csv file was prepared with sample customer data.
  2. The **Data Import Wizard** was used to perform a simple import of the new Contact records.



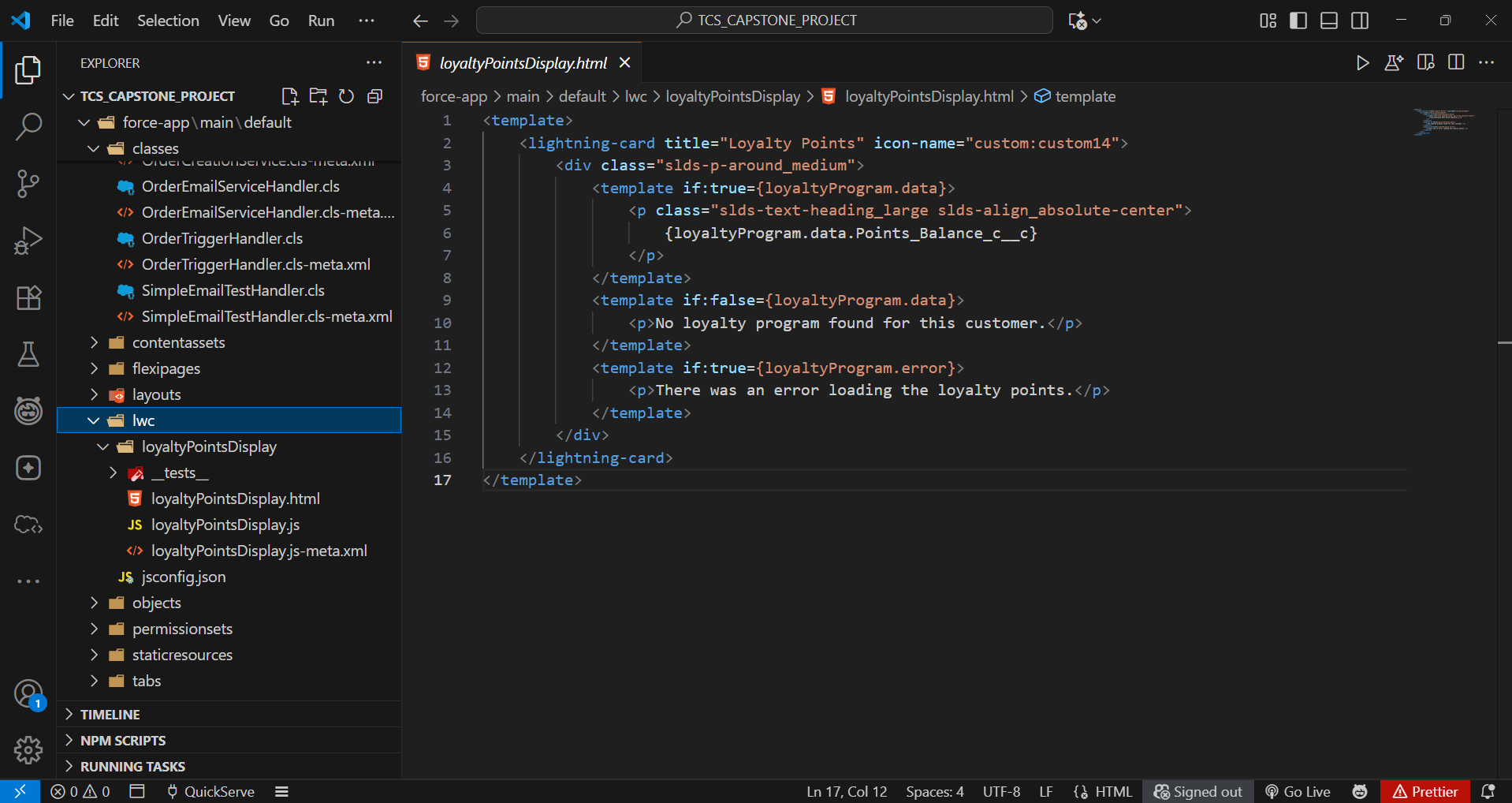
* 1. The new Salesforce IDs for the contacts were exported.
  2. A LoyaltyPrograms.csv file was prepared, using the exported IDs to map the relationship.
  3. The **Data Loader** client application was installed and used to insert the Loyalty\_Program\_\_c records, as it can handle mapping lookup fields on custom objects.



**♦️ Deployment Strategy**

The deployment strategy was defined based on the tools available in the development environment and modern development practices.

* **Change Sets**: It was determined that Change Sets, the standard tool for deploying from a sandbox, are not available in a standalone Developer Edition org because they require a deployment connection.
* **VS Code & SFDX**: The primary method for all development and deployment was the source-driven approach using Visual Studio Code and the Salesforce CLI (SFDX).



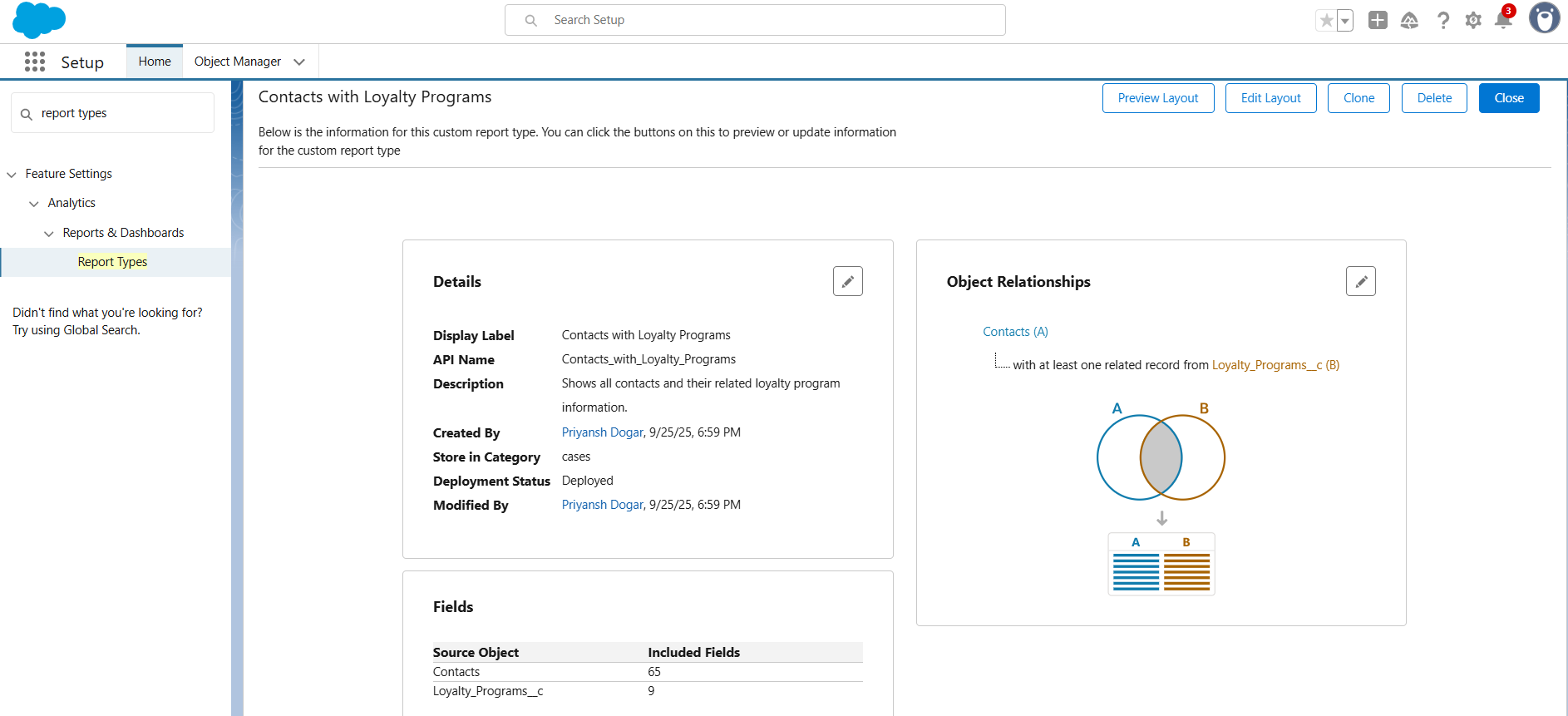
* + **Steps Taken:**
    1. All components (Apex classes, LWCs, Objects, etc.) were created and managed as source files in a local VS Code project.
    2. A version control system (GitHub) was used to track changes.
    3. The SFDX: Deploy Source to Org command was used to migrate all components from the local project to the Salesforce developer org.

**PHASE 9 - Reporting, Dashboards & Security Review**

This phase focuses on visualizing the application's data for analysis and conducting a final review of the security model to ensure data is protected and users have appropriate access.

**♦️ Report Types**

To analyze data across related objects that don't have a standard report type, a custom report type was necessary. This provides the foundation for building new reports that join information from both standard and custom objects.

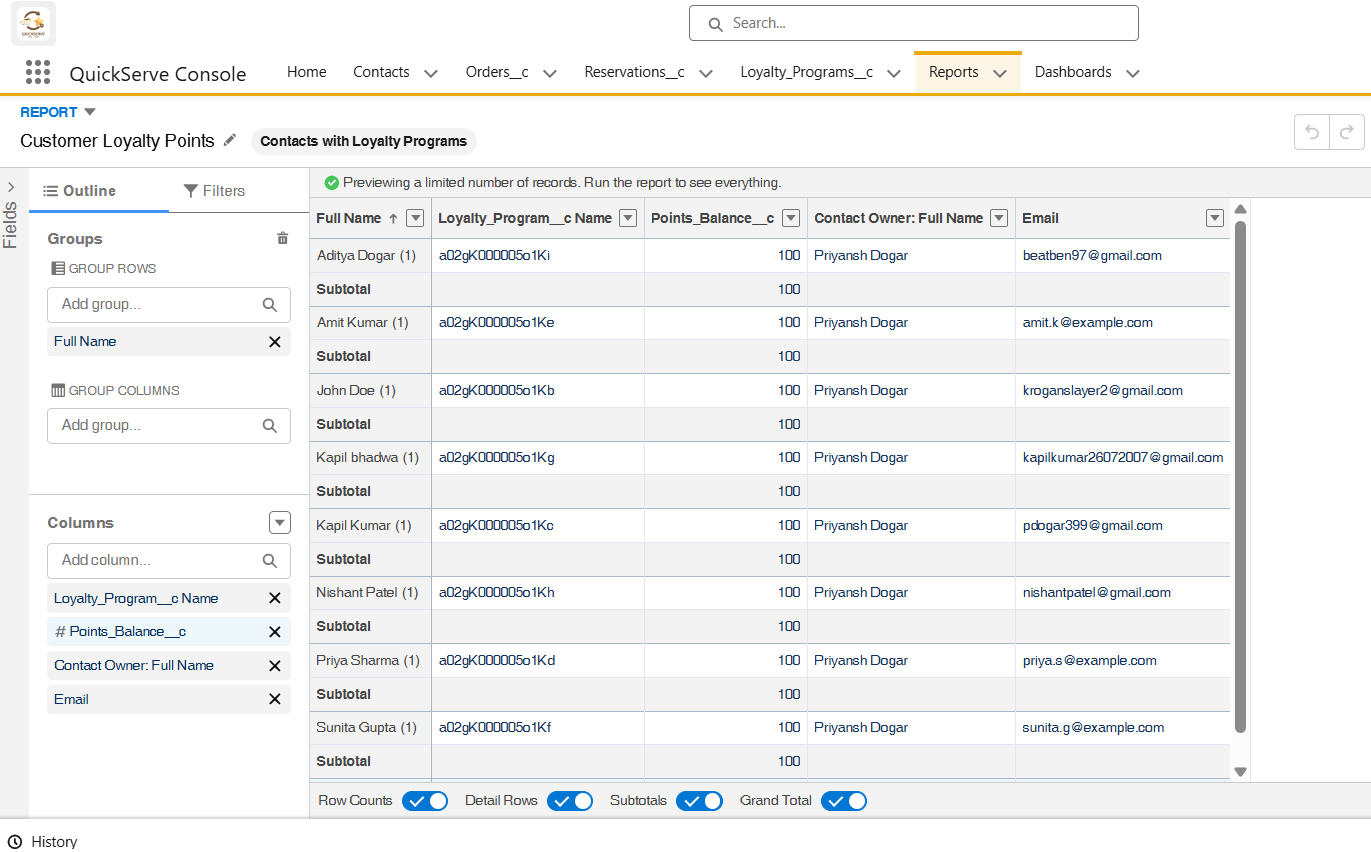


* **Steps Taken:**
  1. A new **Custom Report Type** was created with **Contacts** as the primary object.
  2. The **Loyalty\_Programs\_\_c** object was related to it, with the condition that only contacts with an existing loyalty program record will appear in reports.
  3. The new report type was named "**Contacts with Loyalty Programs**" and saved in a public category for all users to access.

**♦️ Reports**

A new summary report was created to provide managers with a list of all customers in the loyalty program, ranked by their points.

* **Report Name**: Customer Loyalty Points.
* **Report Format**: This was built as a **Summary Report**, which allows for grouping, subtotals, and use in dashboard charts. A simple tabular report would not have been sufficient for visualization.

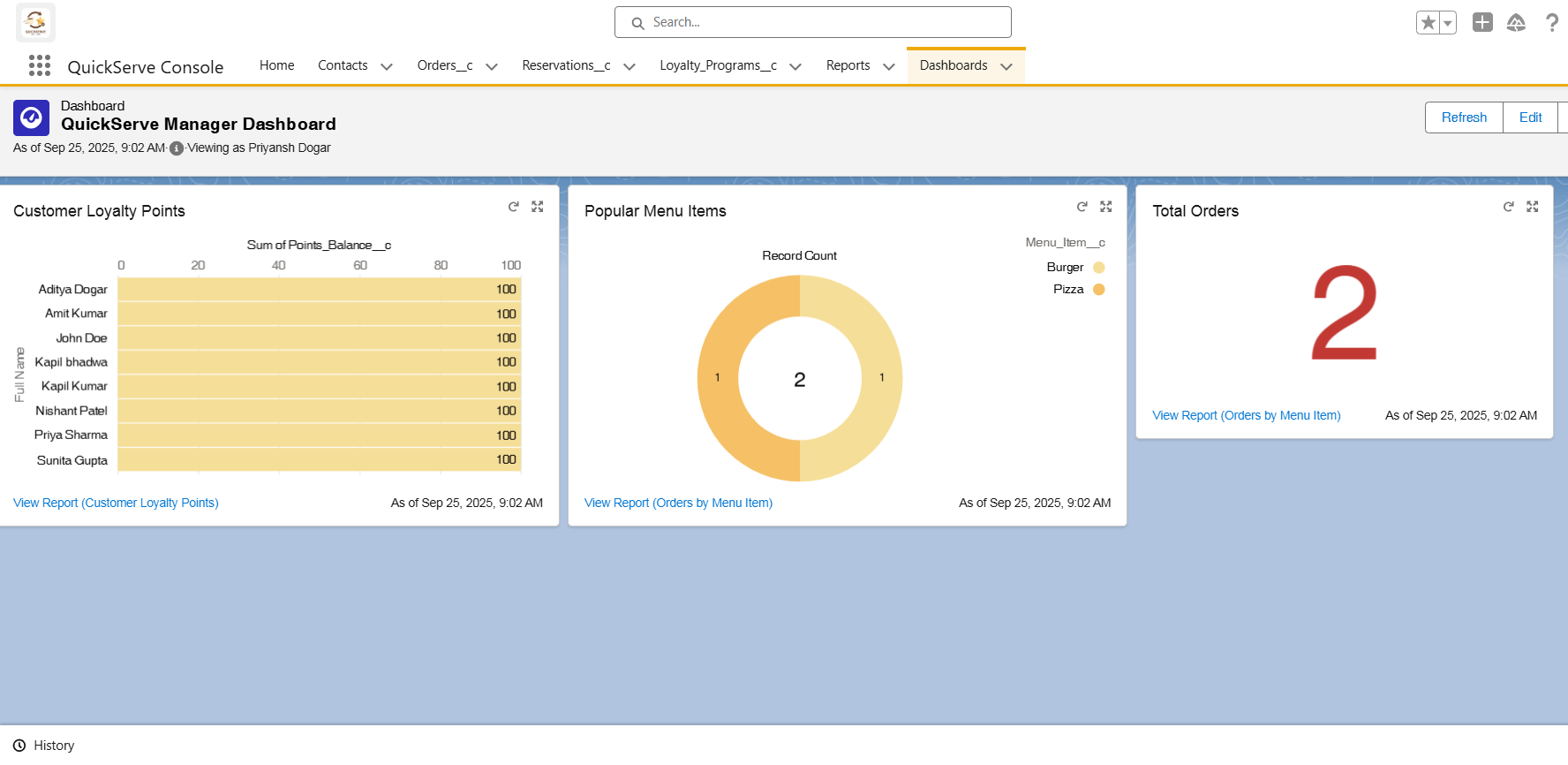


* **Steps Taken:**
  1. A new report was created using the "Contacts with Loyalty Programs" report type.
  2. Key columns were added, including Contact Name, Email, and Points Balance.
  3. The report rows were grouped by the Contact Name field to transform it from a tabular to a summary report.
  4. The final report was saved in a shared public folder so it could be accessed by dashboards and users with the appropriate permissions.

**♦️ Dashboards**

A dashboard was created to give managers a high-level, graphical overview of key performance indicators related to the loyalty program. Dashboards are collections of charts and tables sourced from reports.

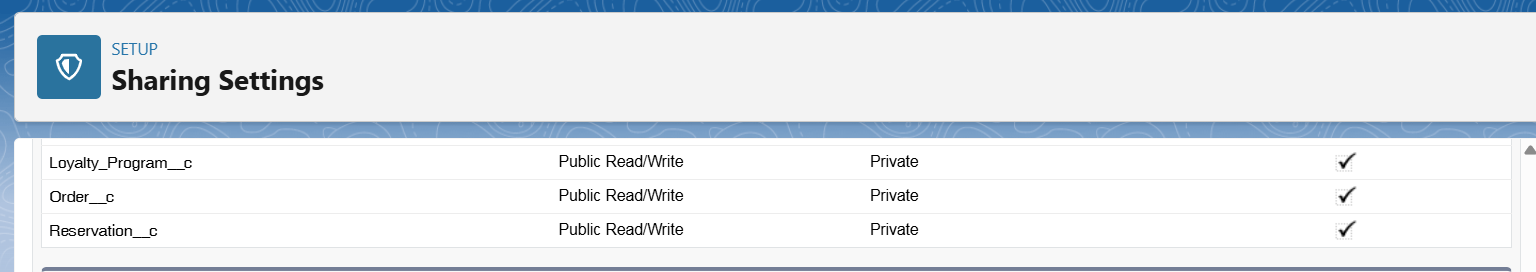
* **Dashboard Name**: QuickServe Manager Dashboard.
* **Components**: A **Horizontal Bar Chart** component was added to the dashboard.
* **Steps Taken:**
  1. A new dashboard was created and saved in a shared folder named "QuickServe Dashboards".
  2. A new component was added, using the Customer Loyalty Points report as its data source.
  3. The component was configured as a Horizontal Bar Chart to create a leaderboard, sorted in descending order by the sum of Points Balance.



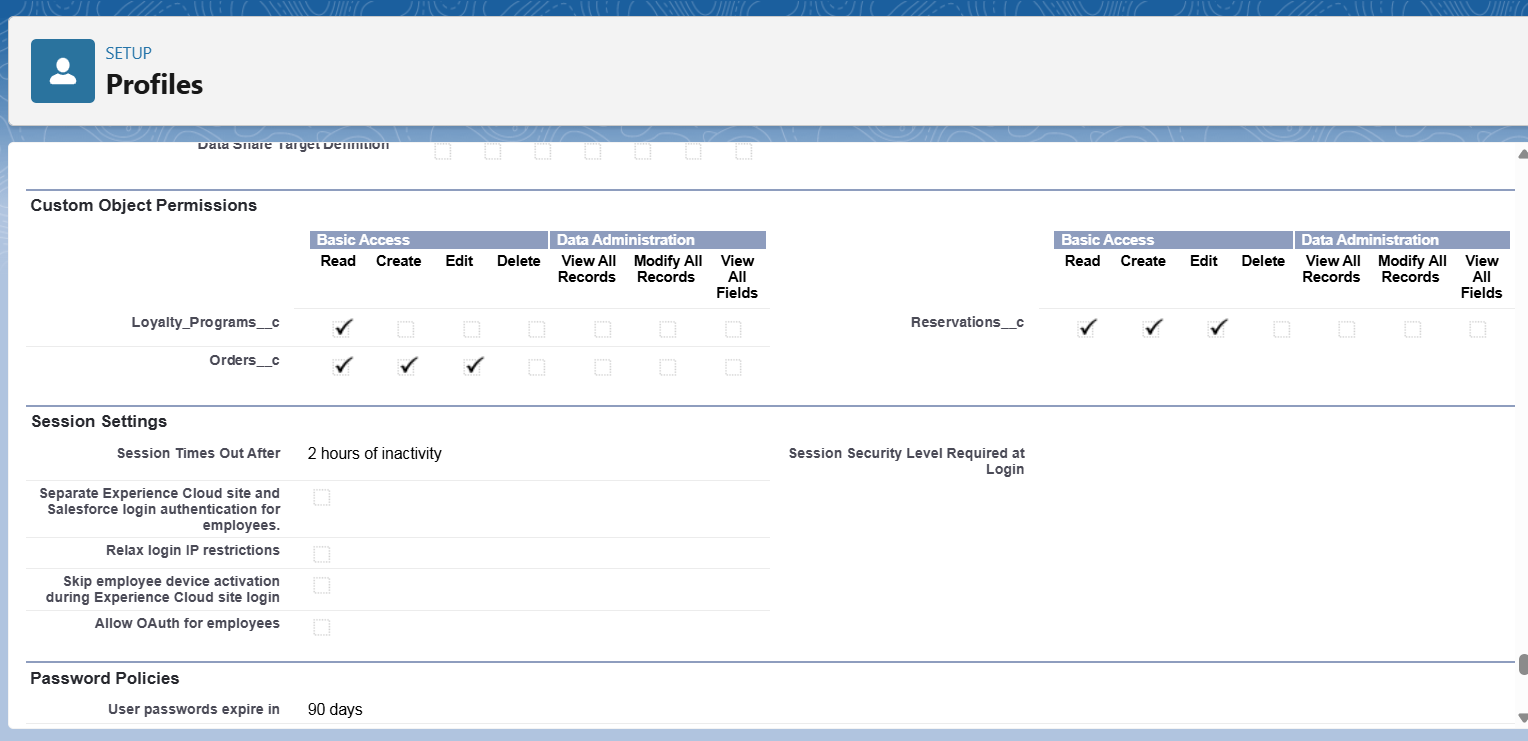
**♦️ Security Review**

A final review of the security settings was conducted to ensure the principle of least privilege was applied across all user profiles.

* **Field Level Security**: The field-level security for the Restaurant\_Staff\_Profile and Manager\_Profile was reviewed and configured to ensure users can only see and edit fields relevant to their roles.
* **Sharing Settings**: The Organization-Wide Defaults and Sharing Rules were reviewed to confirm that data visibility is correctly restricted as configured in earlier phases.



* **Steps Taken:**
  1. Navigated to the Restaurant\_Staff\_Profile and Manager\_Profile in Setup.
  2. Reviewed the "Object Settings" for all custom objects (Order\_\_c, Loyalty\_Program\_\_c, etc.).
  3. Confirmed that the "Basic Access" (Read, Create, Edit, Delete) and "Field Permissions" were set appropriately for each role. For example, staff have read-only access to Points\_Balance\_\_c, while managers have edit access.

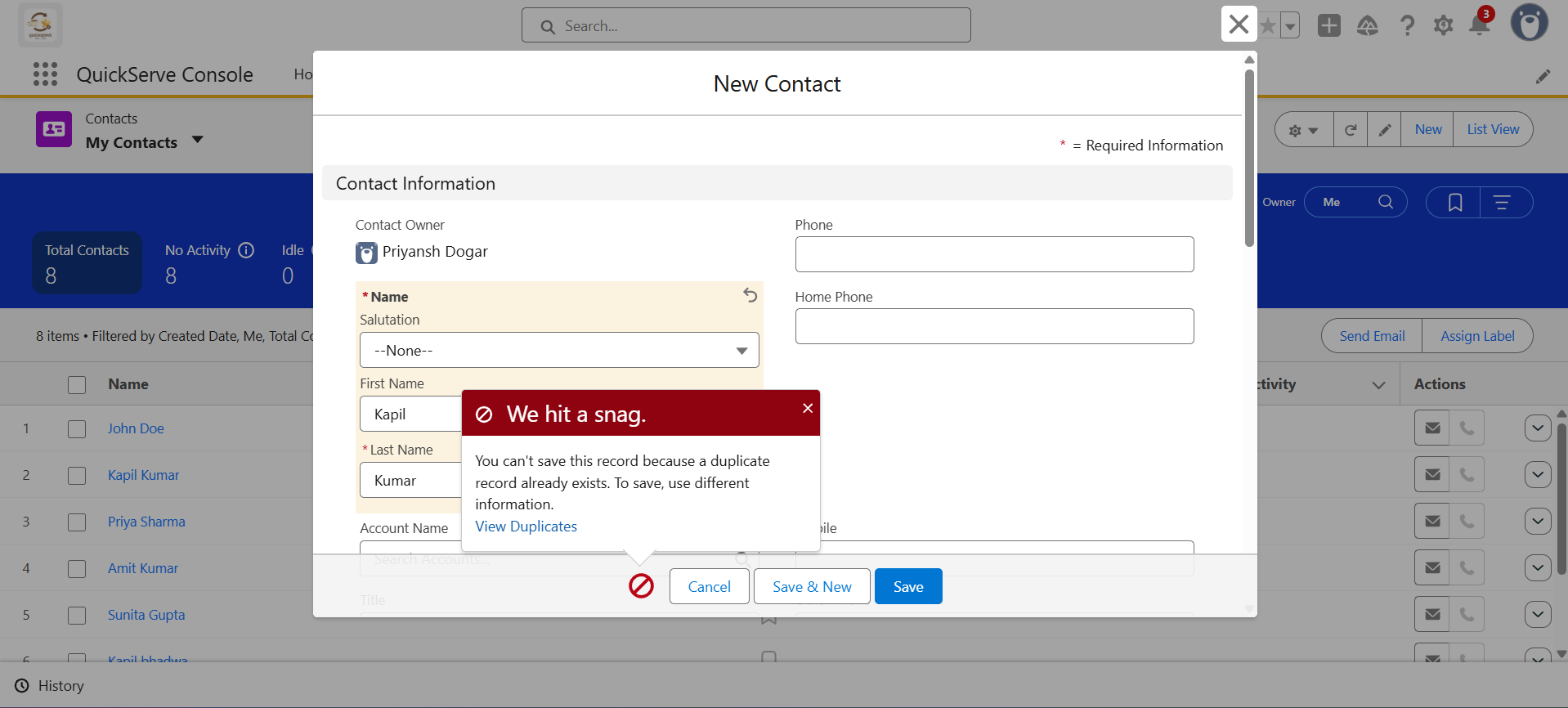


**PHASE 10 - Quality Assurance Testing**

This phase documents the Quality Assurance (QA) testing performed on the key features of the **QuickServe Management System**. Each test case includes the input details, expected output, and a confirmation of the actual output to verify that the implemented features are working as designed.

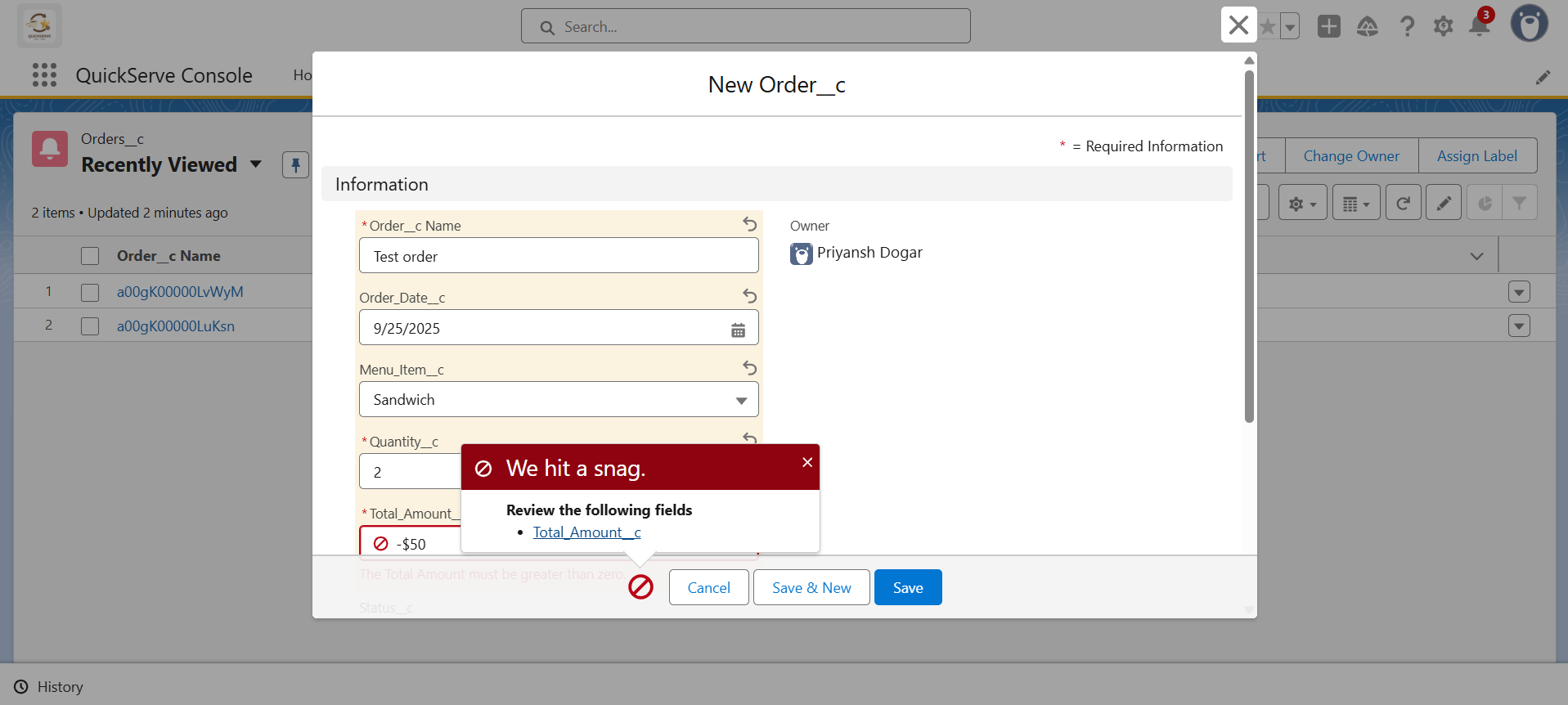
**♦️ Test Case 1: Order Validation Rule**

* **Use Case / Scenario**: Testing the Validation Rule that prevents creating an Order with a zero or negative total amount.
* **Test Steps (with input)**:
  1. Navigated to the Orders tab and clicked "New".
  2. Entered a Total Amount of **-50**.
  3. Clicked "Save".
* **Expected Result**: Salesforce should prevent the record from being saved and display the error message: "The Total Amount must be greater than zero."
* **Actual Result (with Screenshot)**: The actual result matched the expected result. The system correctly blocked the record creation and displayed the specified error message.



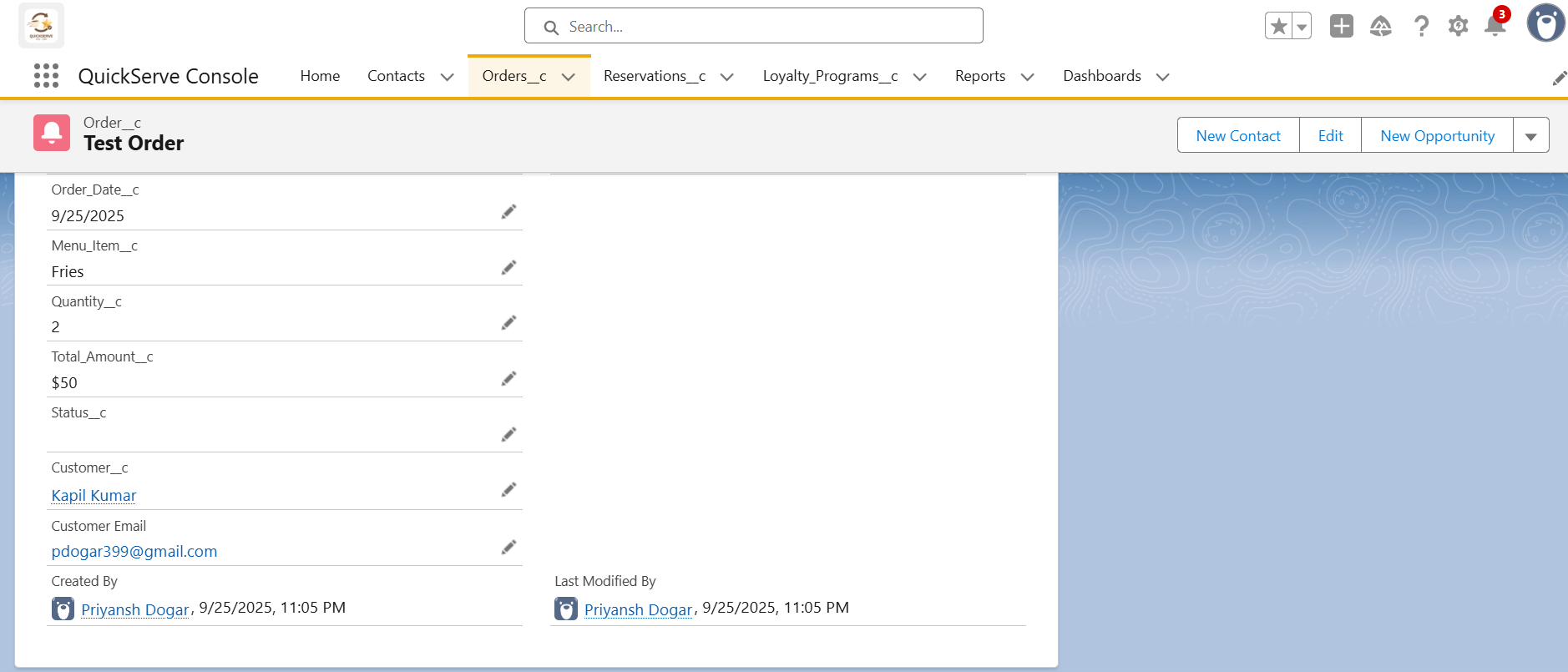
**♦️ Test Case 2: Contact Duplicate Rule**

* **Use Case / Scenario**: Testing the Duplicate Rule that blocks the creation of a new Contact if another contact with the same First Name, Last Name, and Email already exists.
* **Test Steps (with input)**:
  1. Navigated to the Contacts tab and clicked "New".
  2. Entered the First Name, Last Name, and Email of a pre-existing Contact record.
  3. Clicked "Save".
* **Expected Result**: Salesforce should block the record from being saved and display the alert text: "A contact with this name and email already exists."
* **Actual Result (with Screenshot)**: The actual result matched the expected result. The system prevented the duplicate record from being saved and showed the correct error message.



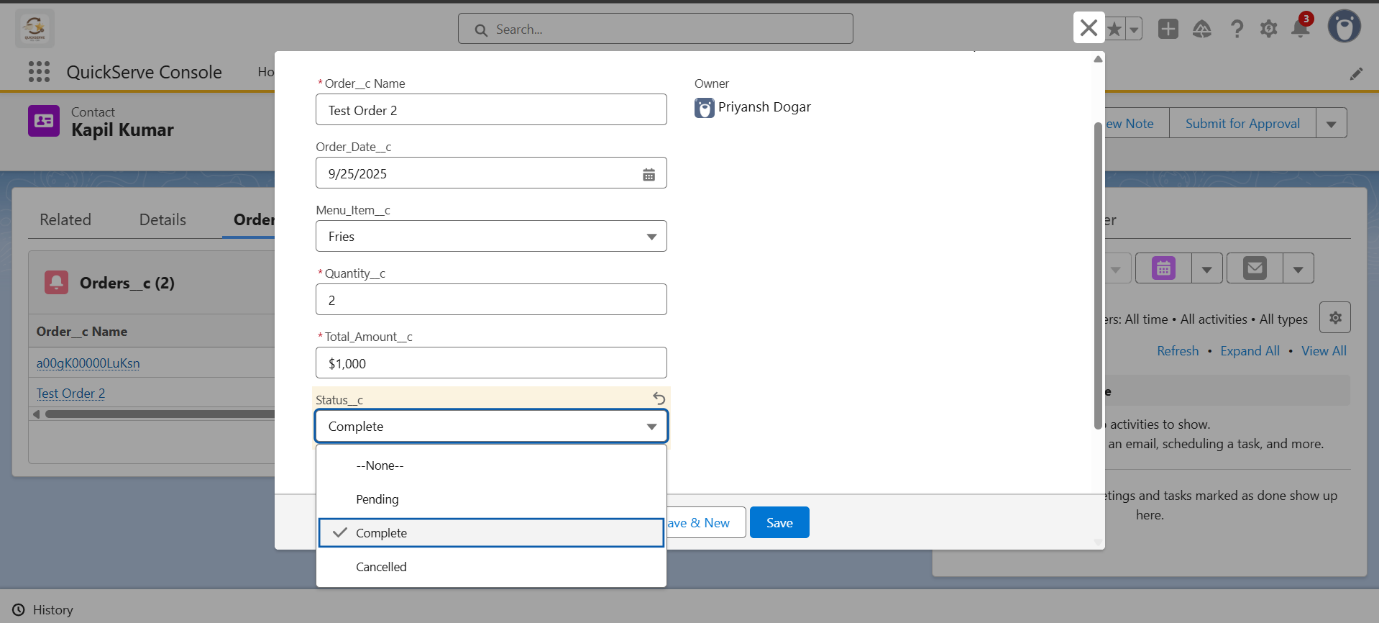
**♦️ Test Case 3: Record-Linking Flow**

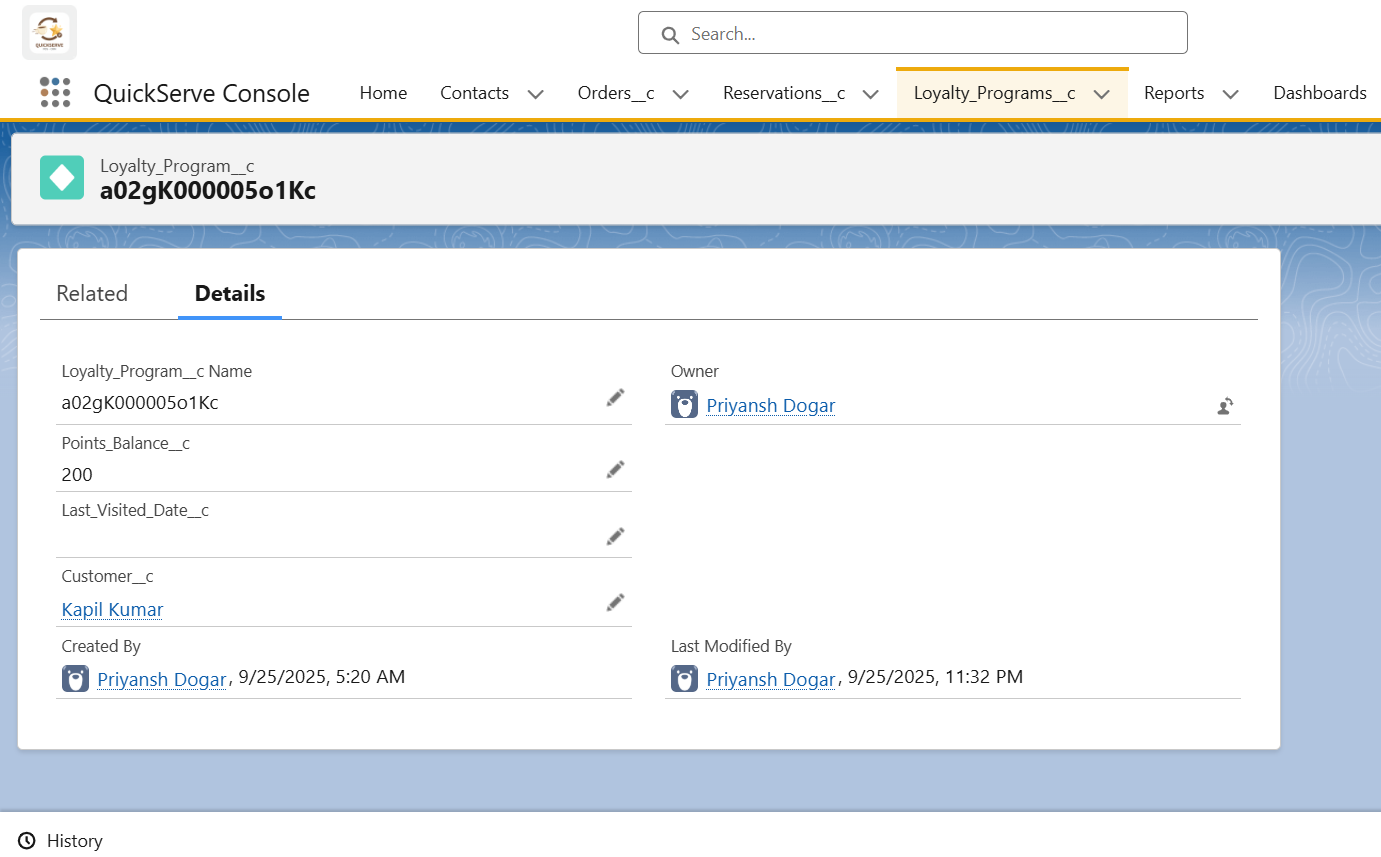
* **Use Case / Scenario**: Testing the record-triggered flow that automatically populates the Customer lookup field on a new Order record based on the value in the Customer Email text field.
* **Test Steps (with input)**:
  1. Created a new Order record.
  2. Left the Customer lookup field blank.
  3. Entered the email of an existing Contact into the Customer Email field.
  4. Saved the record and refreshed the page.
* **Expected Result**: The Customer lookup field on the new Order record should be automatically populated with the correct Contact record.
* **Actual Result (with Screenshot)**: The actual result matched the expected result. The flow correctly identified the contact and populated the lookup field.



**♦️ Test Case 4: Apex Trigger for Loyalty Points**

* **Use Case / Scenario**: Testing the OrderTrigger that automatically calculates and adds loyalty points to a customer's Loyalty\_Program\_\_c record when an Order\_\_c status is changed to 'Complete'.
* **Test Steps (with input)**:
  1. Noted the current Points Balance on a customer's Loyalty\_Program\_\_c record.
  2. Created a new Order for that customer with a Status of 'Pending' and a Total Amount of 1000.
  3. Edited the Order and changed the Status to 'Complete'.
* **Expected Result**: The Points\_Balance\_\_c on the Loyalty\_Program\_\_c record should increase by 100 (10% of the order total).
* **Actual Result (with Screenshot)**: The actual result matched the expected result. The Apex trigger fired successfully and the points balance was updated correctly.





**♦️ Test Case 5: LWC for Displaying Loyalty Points**

* **Use Case / Scenario**: Testing the loyaltyPointsDisplay Lightning Web Component on the Contact record page to ensure it correctly displays the current points balance.
* **Test Steps (with input)**:
  1. Navigated to the Contact record used in the previous test, whose loyalty balance was updated.
  2. Viewed the record page.
* **Expected Result**: The "Loyalty Points" component in the sidebar should display the correct, up-to-date points balance.
* **Actual Result (with Screenshot)**: The actual result matched the expected result. The LWC correctly fetched and displayed the current points balance.

