Purdue University Salesforce Developer

Live Demo Technical Documentation

By: Ryan Woolsey

Version: 1.00

Table of Contents

[Overview 3](#_Toc147858571)

[Design 4](#_Toc147858572)

[High-Level 4](#_Toc147858573)

[DML Events: 4](#_Toc147858574)

[Components 5](#_Toc147858575)

[Lightning Web Component 5](#_Toc147858576)

[Apex Trigger 5](#_Toc147858577)

[Apex Classes 5](#_Toc147858578)

[Test Classes 6](#_Toc147858579)

# Overview

For this demo, we would like you to demonstrate your ability to integrate external databases into Salesforce and utilize this data to populate location information based on Zip Codes into respective address fields. This demo is a good representation of the complexity of work we do for Purdue.

Project Description:

You are tasked with creating a Salesforce solution that allows users to input a Zip Code, and upon submission, the system should retrieve location data from an external database and populate the corresponding address fields in Salesforce.

**Key Requirements:**

1. External Database Integration: You will need to establish a connection with an external database containing location data. You can use a mock database for this demo but explain how you would approach connecting to a real external data source.

2. Custom Object/Fields: Create a custom object in Salesforce to store the Zip Code and related location data (e.g., city, state, country). Design custom fields within this object to hold the data.

3. Visualforce or Lightning Component: Develop a Visualforce page or a Lightning component that allows users to input a Zip Code.

4. Trigger or Process Builder: Implement an automation process (trigger or process builder) that triggers the retrieval of location data from the external database when a Zip Code is entered.

5. Populate Address Fields: Once the data is retrieved, use it to populate the respective address fields (e.g., City, State, Country) in the Salesforce record.

6. Error Handling: Implement error handling to deal with situations where the Zip Code does not exist in the external database or when there are connection issues.

7. User-Friendly UI: Ensure that the user interface is user-friendly and provides feedback on the data retrieval process.

8. Documentation: Prepare a brief documentation that outlines the key steps and components of your solution.

Your demo should effectively demonstrate the seamless integration of external data sources with Salesforce and the automation of populating address fields based on Zip Code entry.

# Design

## High-Level

The approach I took with this project is to build a solution that is scalable, user-friendly, and pleasant to use.

I started by designing the Custom Object that will hold the Zip Code, City, State, and Country data that the integration pulls back from the database. I then found a database that would meet the needs of this project. I chose a free online database of Zip Code and location data that has a free and accessible API to integrate with. This website is called <https://api.zipcodestack.com>.

To fulfill steps 3, 4, and 5, I built a trigger that invokes a couple of methods in a handler class that then enqueues a method using the queueable interface, which invokes an HTTP callout to Zipcodestack. The method returns a JSON string that I deserialize into a key(zip code)-value(object) pair that I can access to update Salesforce records with the required data.

To meet step 6, I have built various methods to handle any errors that may occur, whether they are connection issues with the database, DML errors, or formatting issues that alert the end user of poor input.

The UI I believe is very simple and effective. It communicates a variety of messages and datapoints to the end user as they use the Lightning Web Component. If they successfully pull back data with a zip code, it will let them know and show them the data that was pulled back from the server.

## DML Events:

The Lightning Web Component only creates one record at a time and does so through imperative callouts to Apex. This Apex Handler class invokes the HTTP callout, pulling back the data, and updating the Location Identifier record before alerting them of its success. This is a synchronous callout.

The Trigger updates Location Identifier records if they are created with a Zip Code filled in, or are updated with the zip code changing. This trigger supports bulk operations. The database only allows 100 zip codes to be queried at a time, so I wrote in logic that breaks any bulk transactions greater than 100 records into multiple chunks.

# Components

## Lightning Web Component

**zipIntegrationLWC\_EnterZip**

Component can be placed on the Home Page or Record Pages.

Consists of an input field for a zip code (string), and a button that invokes the apex class **ZipIntegrationLWCHandler**.

When data is returned, populates a lightning-record-view-form with the Zip, City, State and Country fields.

If component is on a record page (Account, Contact, Lead), will also update the related record. This is handled by a method on the handler class.

## Apex Trigger

**LocationIdentifierTrigger**

* Invokes the **ZipIntegrationLocationIdentifierHandler** class on after insert (Trigger.new) and after update (Trigger.newMap, Trigger.oldMap) DML operations. All logic is handled in handler class.

## Apex Classes

**ZipIntegrationLocationIdentifierHandler**

* Contains three methods: handleAfterUpdate, handleAfterInsert, and handleBulkJobs.
* handleAfterUpdate uses the new and old maps to compare and determine if the Zip code has changed. If so, will add the Id of the Location Identifier record and the Zip Code in a map which is enqueued in a queueable method, found in the **ZipIntegrationHTTPHandler** class.
* handleAfterInsert uses a list of record in the trigger and adds them to a map just like afterUpdate. Invokes the same queueable class with the map as an argument.

**ZipIntegrationHTTPHandler**

* Implements Queueable and Database.AllowsCallouts intefaces. Accepts a Map of a String key and String value pair as an argument. Builds a single string of all Zip Codes found in comma separated format and invokes the **processPayload** method, which creates the HTTP Request. This method returns a JSON String, which is then processed by the **ZipIntegrationJSONUntypedHandler**.

**ZipIntegrationJSONUntypedHandler**

* Processes the JSON string to extract the data from Zipcodestack.com.
* This method invokes a second method called assignLocationIdVariables which returns a Location Identifier record. This is then added to a list and returned to the HTTPHandler class. It then uses a try catch block to insert the records, handled by an exception method if errors occur.

**ZipIntegrationLWCHandler**

* This class is invoked imperatively by the Lightning Web Component, zipIntegrationLWC\_EnterZip. It accepts a zip code and a record Id as arguments and returns a string that is used to alert the end user of its result (success or fail).
* This class invokes the same HTTP methods synchronously that the Queueable interface invoked asynchronously. This allows immediate response to the end user on their data.
* It utilizes a method called **validateRecord** that validates that a Zip code was found and alerts the end user on whether it was or not. If it was, it will continue to create the new Location Identifier record and update the Lead, Contact or Account record, if on one of those record pages.
* If this is the case, a void method is invoked called updateObjectRecord that accepts the record Id of the record to be updated, and the Location Identifier record. This record is used to populate the Account, Contact or Lead City, State and Country fields.

## Test Classes

**ZIpIntegrationTest**

* Contains 12 methods. One is a @testSetup method to make data for testing purposes, 10 are @isTest methods to test various functionality, and the final method generates HTTP Mock data. All tests combined generate 92% coverage at the time of writing this document.

**ZipIntegrationHTTPMockTest**

* Creates a list of HTTPResponse records that can be created dynamically in Test Classes.