# Salesforce Developer Superset

**Skills Required:**  
Salesforce Developer

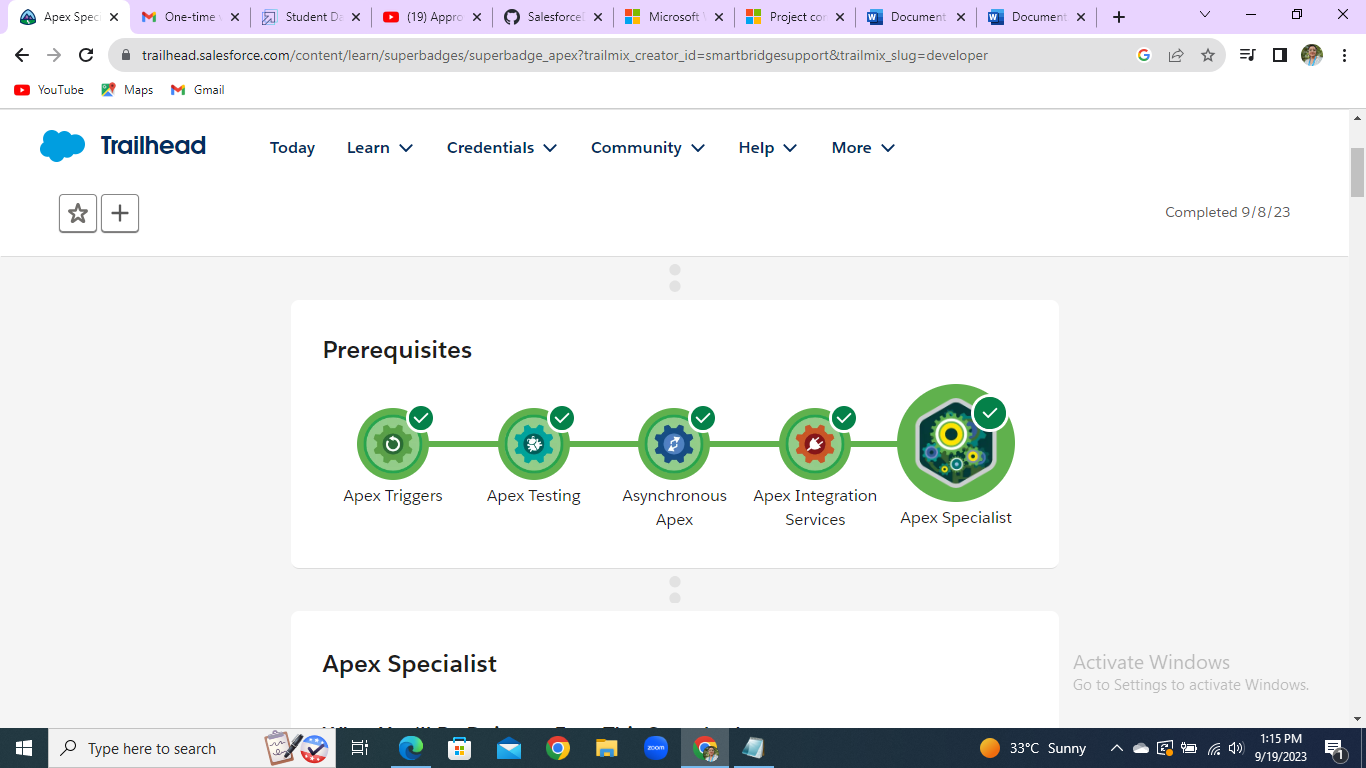
**Project Description:**

The Developer Super Set capstone is a Salesforce credential made up of two superbadges.

* Apex Specialist
* Process Automation Specialist

Complete these superbadges to unlock this super set capstone, then complete the challenges below to earn the Developer Super Set.

Apex Specialist Super Superbadge



**Steps to complete this superbadge:**

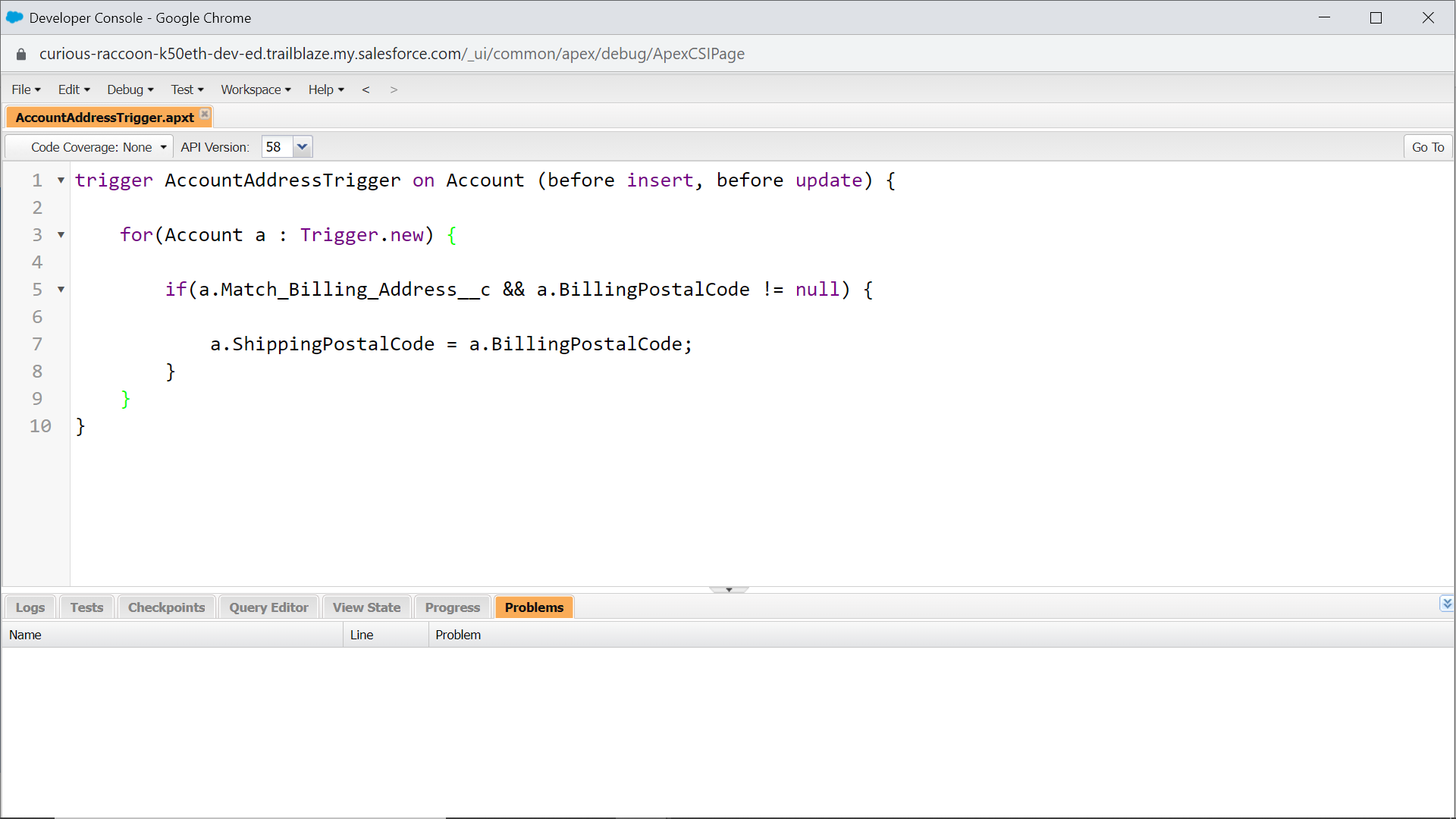
* Automate record creation using Apex triggers
* Synchronize Salesforce data with an external system using asynchronous REST callouts
* Schedule synchronization using Apex code
* Test automation logic to confirm Apex trigger side effects
* Test integration logic using callout mocks
* Test scheduling logic to confirm action gets queued
* Apex Triggers Module:

**Create an Apex trigger**

Creating an Apex trigger that sets an account’s Shipping Postal Code to match the Billing Postal Code if the Match Billing Address option is selected. Fire the trigger before inserting an account or updating an account.

**Pre-Work:**  
**Adding a checkbox field to the Account object:**

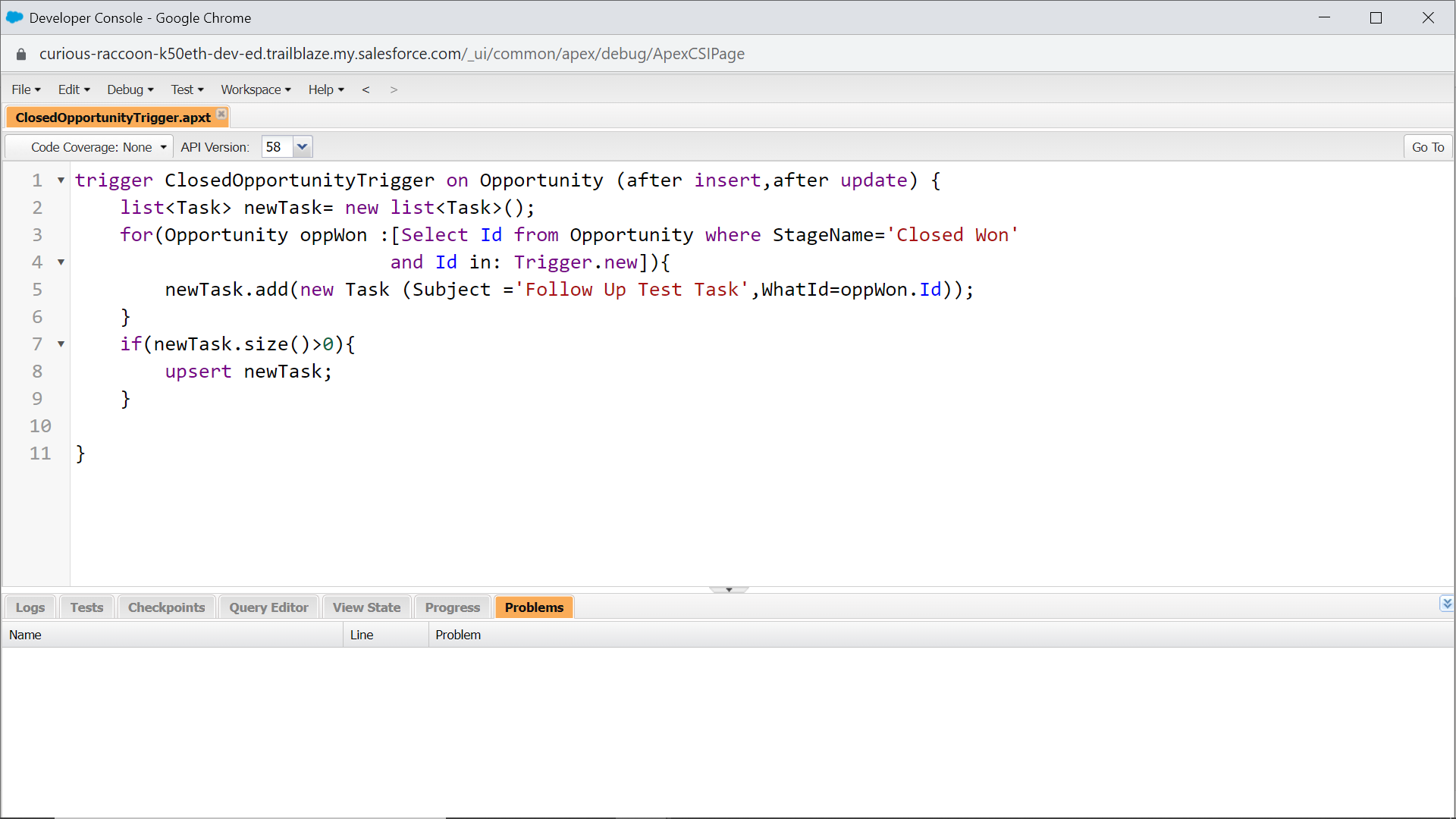
* Field Label: Match Billing Address
* Field Name: Match\_Billing\_Address  
  Note: The resulting API Name should be Match\_Billing\_Address\_\_c.
* Name: AccountAddressTrigger
* Object: Account
* Events: before insert and before update
* Condition: Match Billing Address is true
* Operation: setting the Shipping Postal Code to match the Billing Postal Code



**Creating a Bulk Apex trigger**

Creating a bulkified Apex trigger that adds a follow-up task to an opportunity if its stage is Closed Won. Fire the Apex trigger after inserting or updating an opportunity.

* Create an Apex trigger:
* Name: ClosedOpportunityTrigger
* Object: Opportunity
* Events: after insert and after update
* Condition: Stage is Closed Won
* Operation: Create a task:
* Subject: Follow Up Test Task
* WhatId: the opportunity ID (associates the task with the opportunity)
* Bulkifying the Apex trigger so that it can insert or update 200 or more opportunities



## Apex Testing Module:

**Creating a Unit Test for a Simple Apex Class**

Creating and installing a simple Apex class to test if a date is within a proper range, and if not, returns a date that occurs at the end of the month within the range. You'll copy the code for the class from GitHub. Then write unit tests that achieve 100% code coverage.

* Create an Apex class:
* Name: VerifyDate
* Code: [**Copy from GitHub{code copied from GitHub}**](https://github.com/developerforce/trailhead-code-samples/blob/master/VerifyDate.cls)
* Place the unit tests in a separate test class:
* Name: TestVerifyDate
* Goal: 100% code coverage

100% code coverage obtained.

**Creating a Unit Test for a Simple Apex Trigger**

Creating and installing a simple Apex trigger which blocks inserts and updates to any contact with the last name of 'INVALIDNAME'. You'll copy the code for the class from GitHub. Then write unit tests that achieve 100% code coverage.

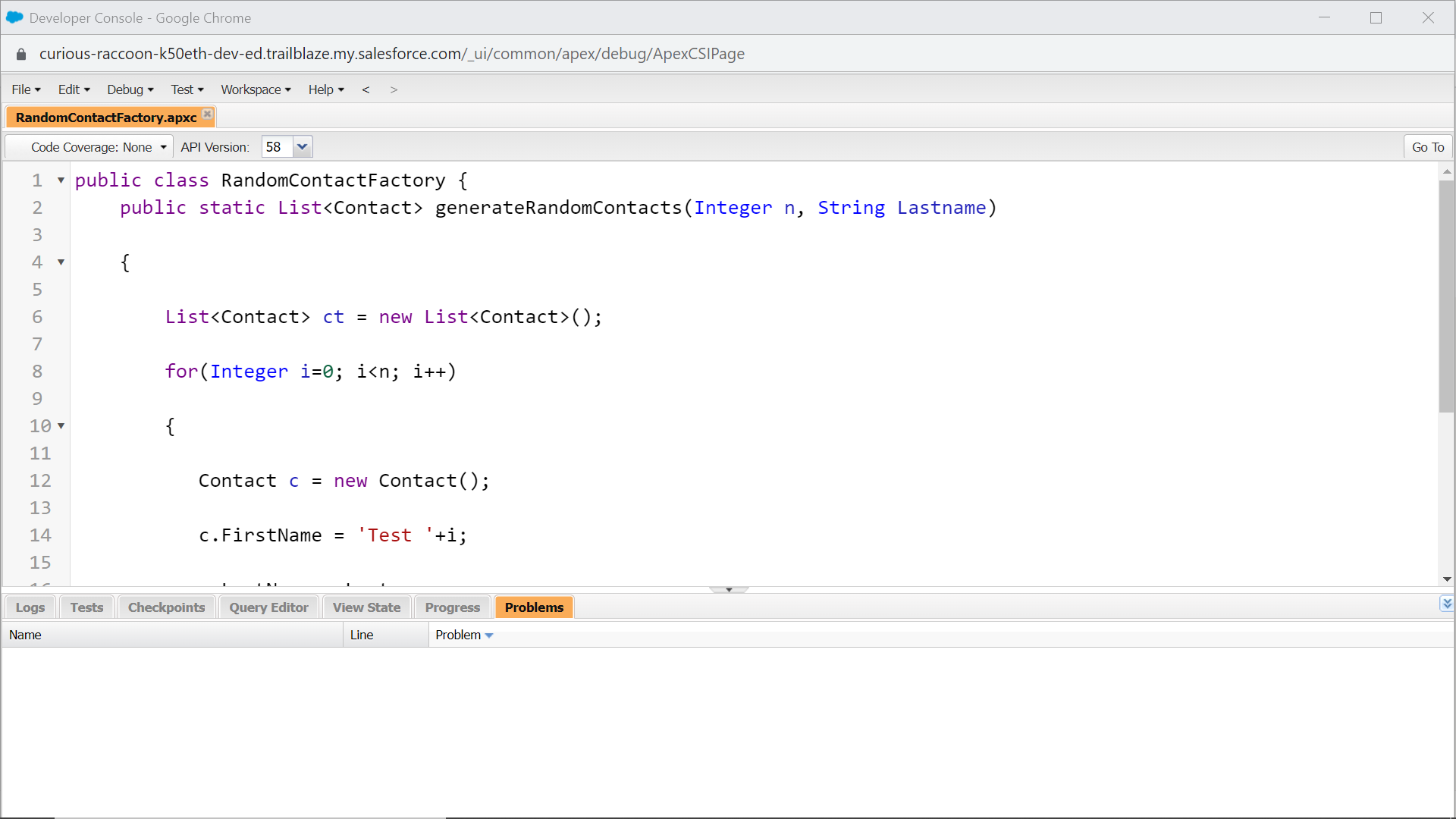
* Create an Apex trigger on the Contact object
* Name: RestrictContactByName
* Code: [**Copy from GitHub**](https://github.com/developerforce/trailhead-code-samples/blob/master/RestrictContactByName.cls)
* Place the unit tests in a separate test class
* Name: TestRestrictContactByName
* Goal: 100% test coverage

100% code coverage obtained.

**Creating a Contact Test Factory**

Creating an Apex class that returns a list of contacts based on two incoming parameters: the number of contacts to generate and the last name. Do not insert the generated contact records into the database.

* Creating an Apex class in the public scope
* Name: RandomContactFactory (without the @isTest annotation)
* Using a Public Static Method to consistently generate contacts with unique first names based on the iterated number in the format Test 1, Test 2 and so on.
* Method Name: generateRandomContacts (without the @isTest annotation)
* Parameter 1: An integer that controls the number of contacts being generated with unique first names
* Parameter 2: A string containing the last name of the contacts
* Return Type: List < Contact >



## Asynchronous Apex :

**Creating an Apex class that uses the @future annotation to update Account records.**

Creating an Apex class with a future method that accepts a List of Account IDs and updates a custom field on the Account object with the number of contacts associated to the Account.

* Creating a field on the Account object:
* Label: Number Of Contacts
* Name: Number\_Of\_Contacts
* Type: **Number**
* This field will hold the total number of Contacts for the Account
* Create an Apex class:
* Name: AccountProcessor
* Method name: countContacts
* The method must accept a List of Account IDs
* The method must use the @future annotation
* The method counts the number of Contact records associated to each Account ID passed to the method and updates the 'Number\_Of\_Contacts\_\_c' field with this value
* Create an Apex test class:
* Name: AccountProcessorTest

100% code coverage obtained.

**Create an Apex class that uses Batch Apex to update Lead records.**

Create an Apex class that implements the Database.Batchable interface to update all Lead records in the org with a specific LeadSource.

* Create an Apex class:
* Name: LeadProcessor
* Interface: Database.Batchable
* Use a QueryLocator in the start method to collect all Lead records in the org
* The execute method must update all Lead records in the org with the LeadSource value of Dreamforce
* Create an Apex test class:
* Name: LeadProcessorTest
* In the test class, insert 200 Lead records, execute the LeadProcessor Batch class and test that all Lead records were updated correctly

100% code coverage obtained.

**Creating a Queueable Apex class that inserts Contacts for Accounts.**

Creating a Queueable Apex class that inserts the same Contact for each Account for a specific state.

* Create an Apex class:
* Name: AddPrimaryContact
* Interface: Queueable
* Create a constructor for the class that accepts as its first argument a Contact sObject and a second argument as a string for the State abbreviation
* The execute method must query for a maximum of 200 Accounts with the BillingState specified by the State abbreviation passed into the constructor and insert the Contact sObject record associated to each Account. Look at the sObject clone() method.
* Create an Apex test class:
* Name: AddPrimaryContactTest
* In the test class, insert 50 Account records for BillingState NY and 50 Account records for BillingState CA
* Create an instance of the AddPrimaryContact class, enqueue the job, and assert that a Contact record was inserted for each of the 50 Accounts with the BillingState of CA

100% code coverage obtained.

**Creating an Apex class that uses Scheduled Apex to update Lead records.**

Creating an Apex class that implements the Schedulable interface to update Lead records with a specific LeadSource.

* Create an Apex class:
* Name: DailyLeadProcessor
* Interface: Schedulable
* The execute method must find the first 200 Lead records with a blank LeadSource field and update them with the LeadSource value of Dreamforce
* Create an Apex test class:
* Name: DailyLeadProcessorTest
* In the test class, insert 200 Lead records, schedule the DailyLeadProcessor class to run and test that all Lead records were updated correctly

100% code coverage obtained.

APEX SPECIALIST

**Use Case:**

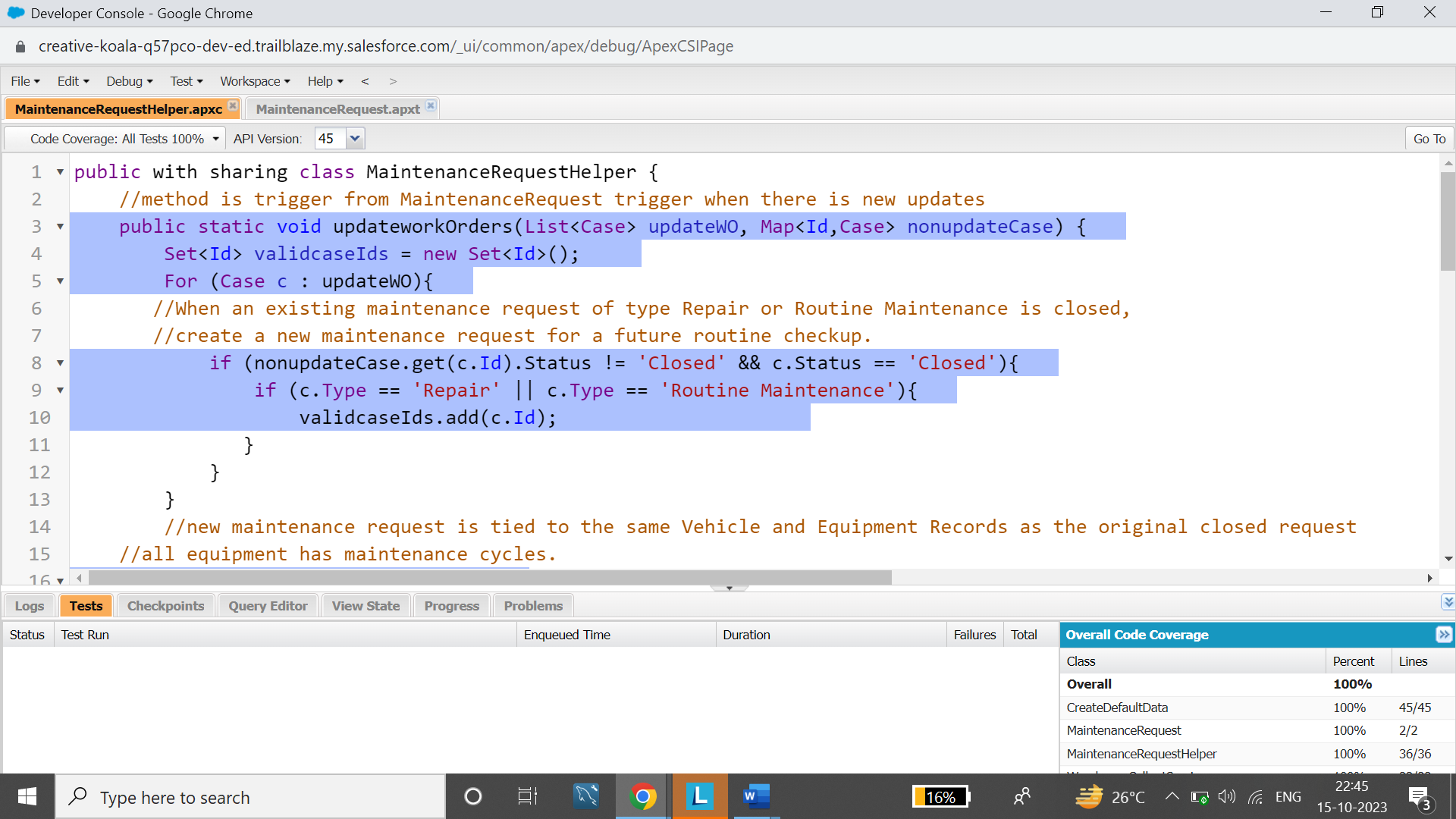
The RV community is increasing exponentially across the globe. Over the past few years, HowWeRoll Rentals, the world’s largest RV rental company, has increased its global footprint and camper fleet tenfold. HowWeRoll offers travelers superior RV rental and roadside assistance services.

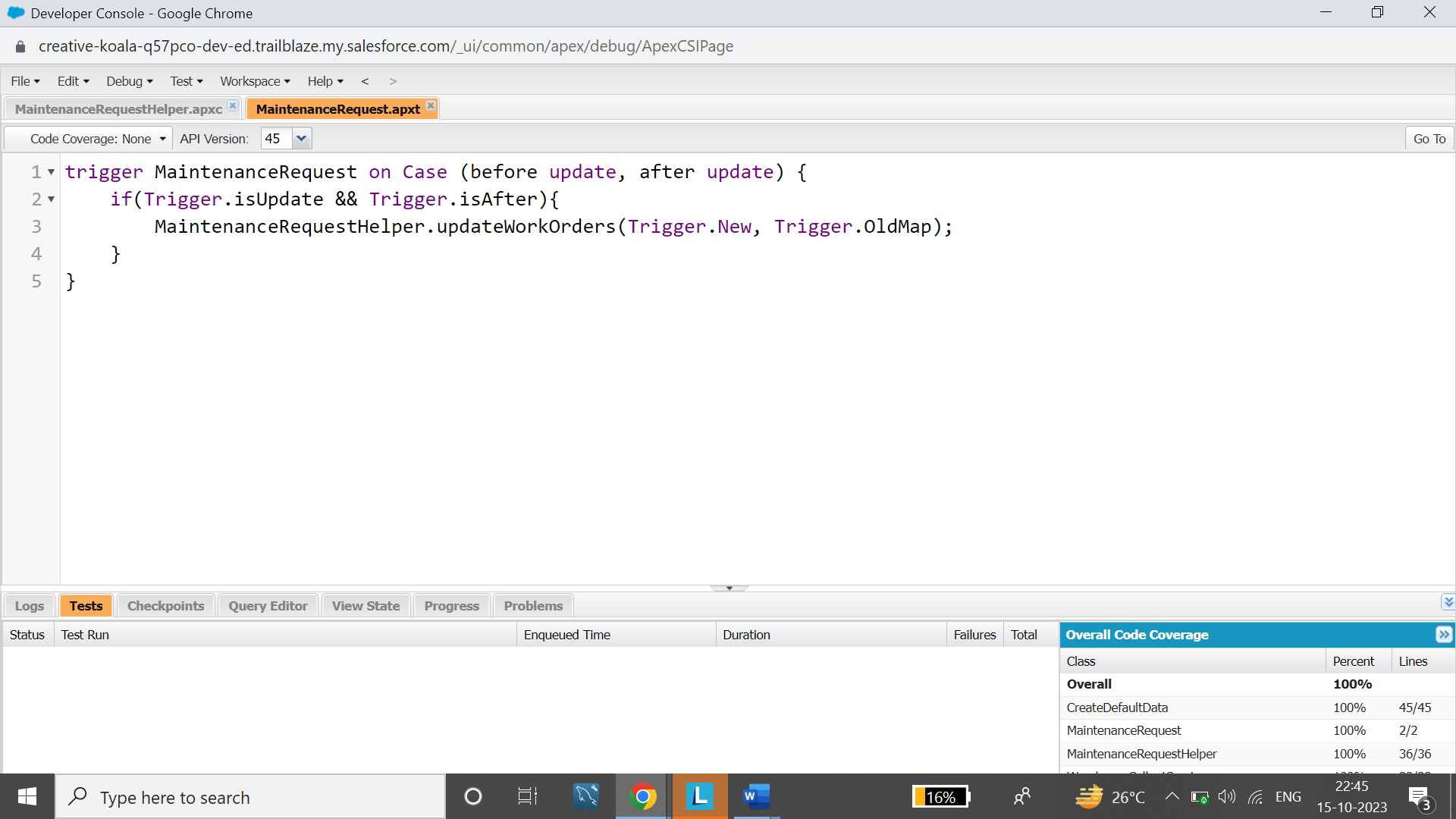
You have been hired as the lead Salesforce developer to automate and scale HowWeRoll’s reach. For travelers, not every journey goes according to plan. Unfortunately, there’s bound to be a bump in the road at some point along the way. Thankfully, HowWeRoll has an amazing RV repair squad who can attend to any maintenance request, no matter where you are. These repairs address a variety of technical difficulties, from a broken axle to a clogged septic tank.

As the company grows, so does HowWeRoll’s rental fleet. While it’s great for business, the current service and maintenance process is challenging to scale. In addition to service requests for broken or malfunctioning equipment, routine maintenance requests for vehicles have grown exponentially. Without routine maintenance checks, the rental fleet is susceptible to avoidable breakdowns.

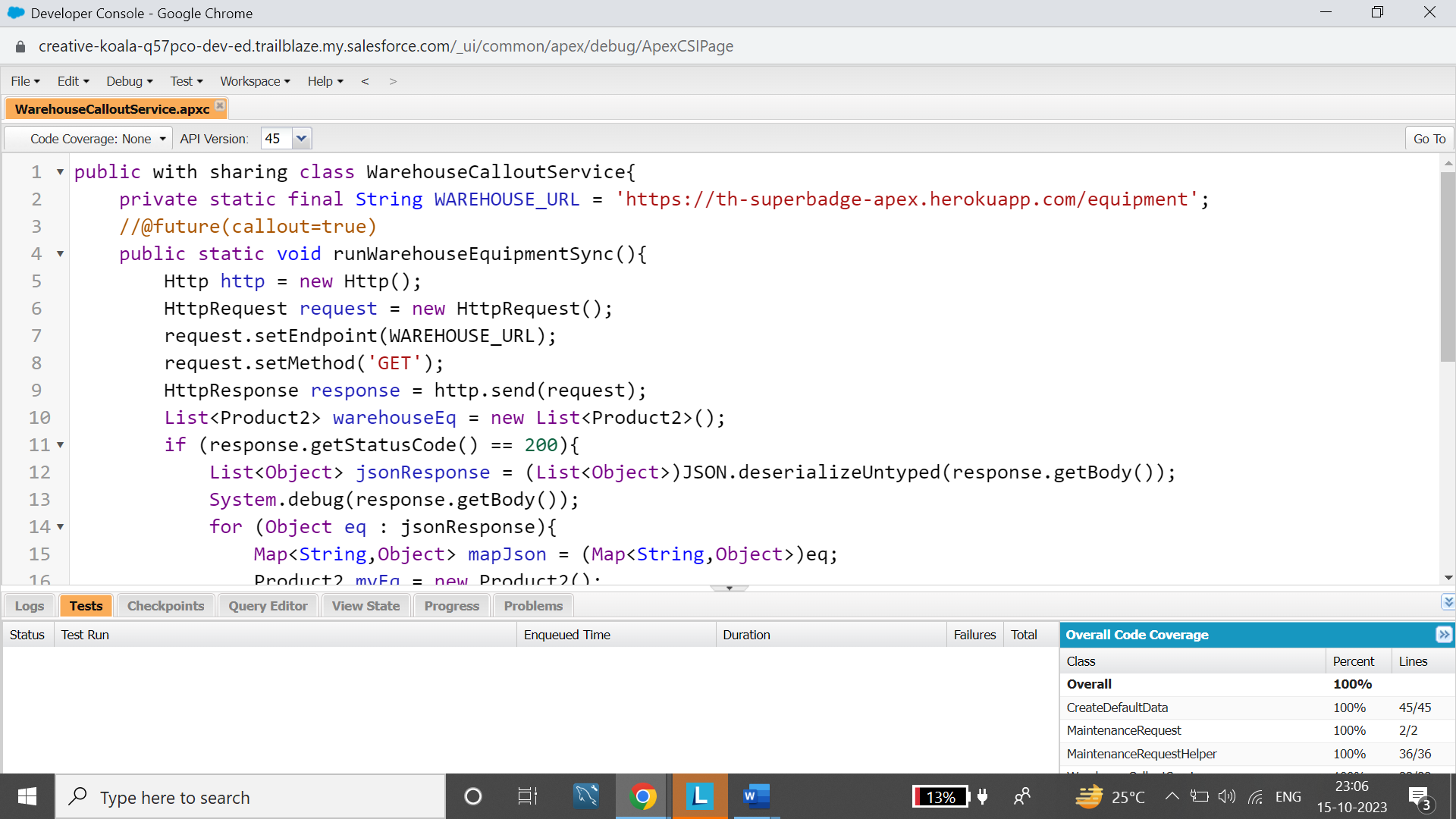
That’s where you come in! HowWeRoll needs you to automate their Salesforce-based routine maintenance system. You’ll ensure that anything that might cause unnecessary damage to the vehicle, or worse, endanger the customer is flagged. You’ll also integrate Salesforce with HowWeRoll’s back-office system that keeps track of warehouse inventory

#### **Automate Maintenance Requests**

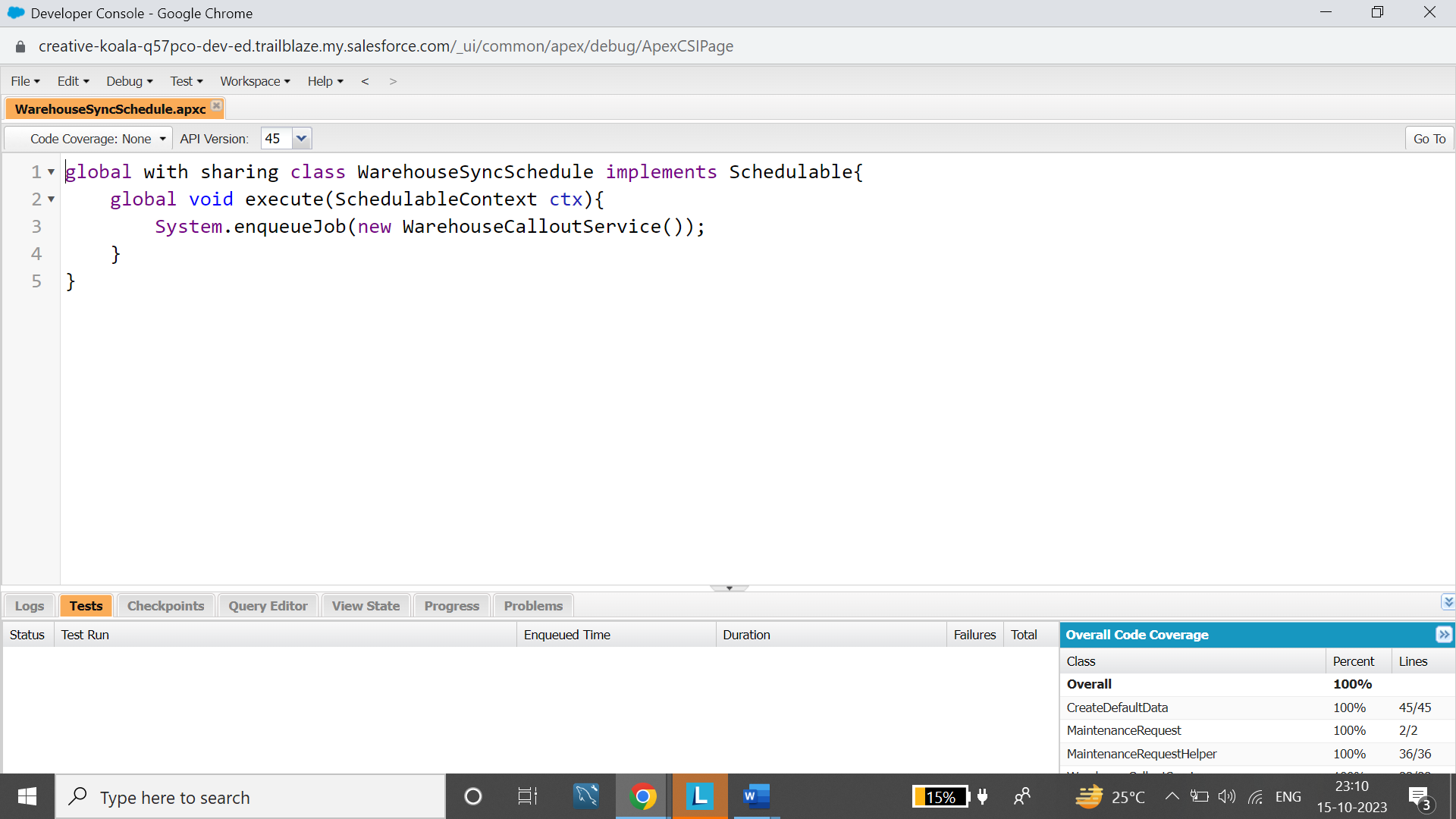




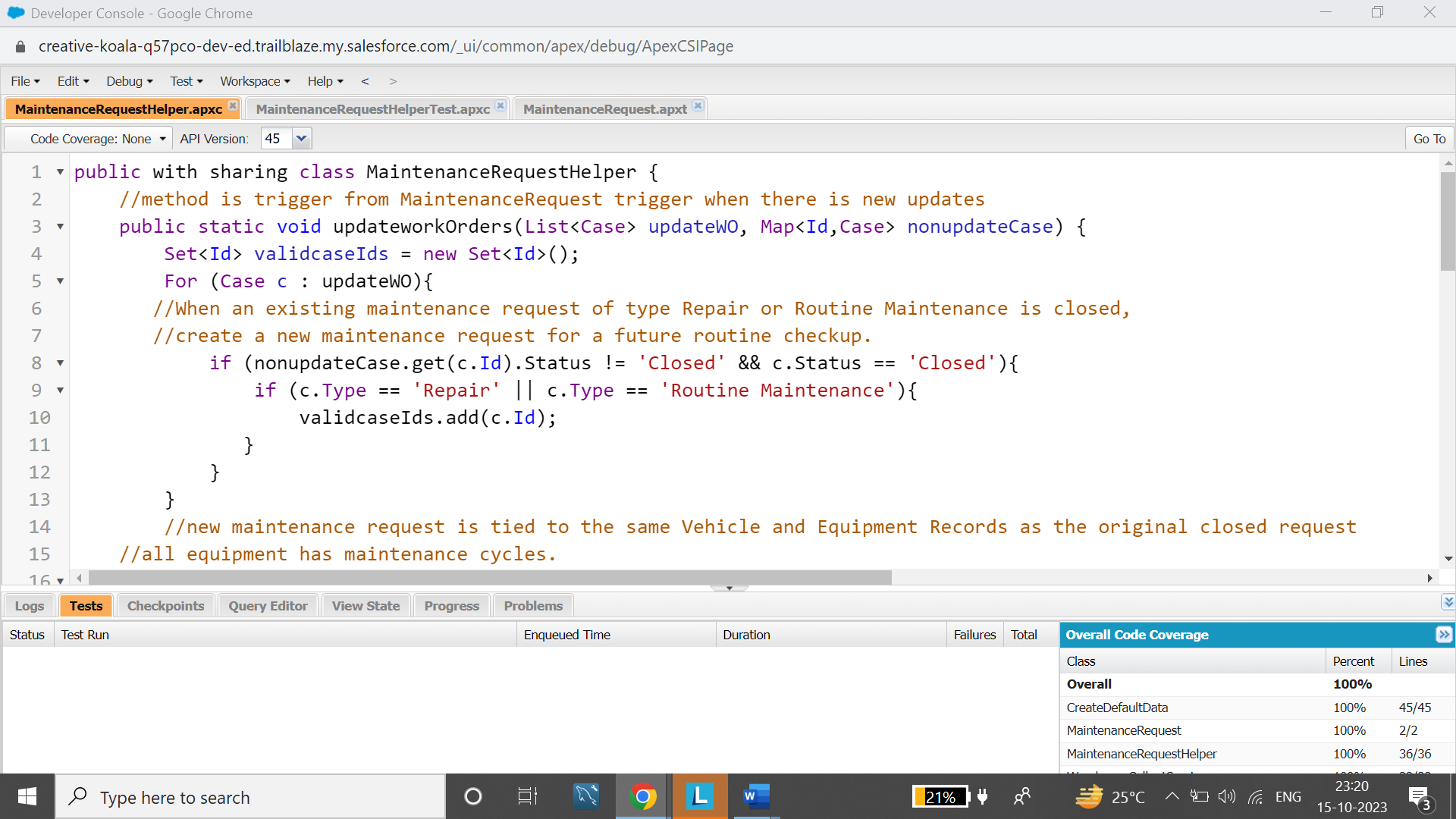
**Synchronize Salesforce Data**

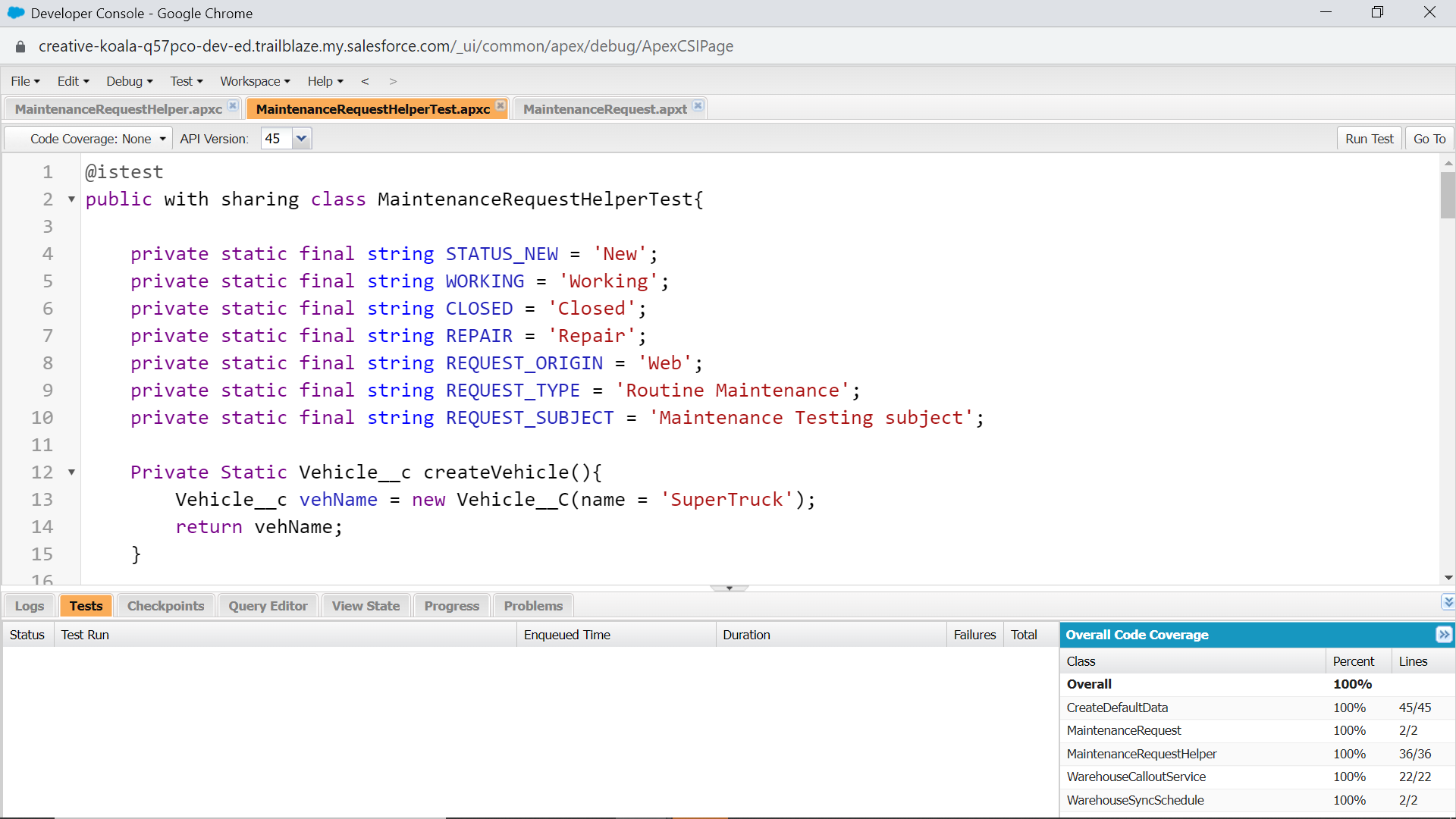


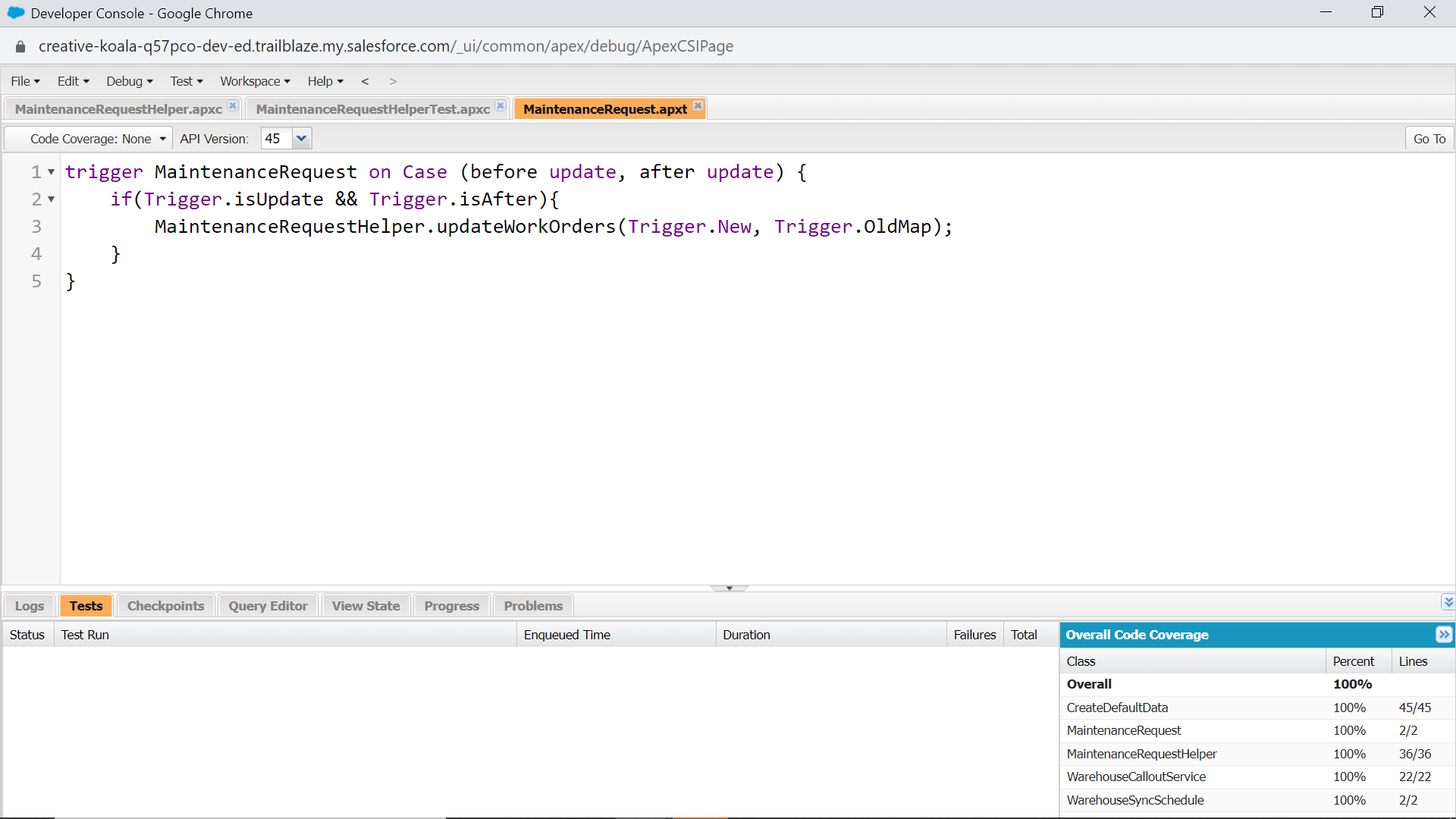
**Schedule Synchronization**



**Test Automation Logic**







**Test Callout Logic**

