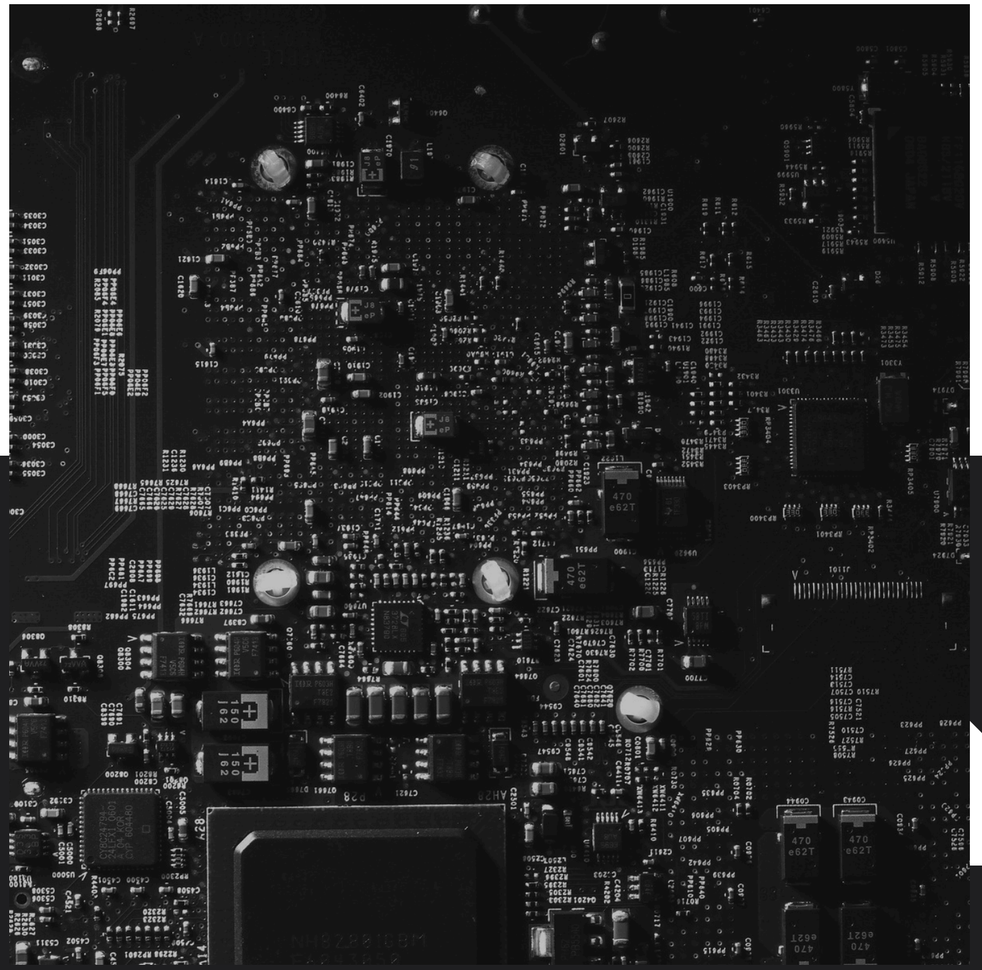
CAPSTONE PROJECT -

MULTI FRAMEWORK

WEB AUTOMATION &BACKEND ENGINEERING

Presented by Group 04



Objective

To build a synchronized ecosystem where automation frameworks and backend logic were developed in parallel. We focused on maintaining code consistency and peer-reviewing logic to ensure a bug-free delivery across all three projects.

Our Team

Anmol Singhal
Trainee

Nalamalapu Mohan Reddy
Trainee

Aditya Raghuwanshi
Trainee

Rahul Veerabathini
Trainee

OUR PROJECTS

01

TOOLSHOP AUTOMATION PYTEST:
A high-performance automation framework built with Pytest and Selenium to validate complex user journeys, including dynamic cart management and mathematical price verification.

02

Robot Framework (Keyword-Driven):
A modular testing suite leveraging Robot Framework to abstract technical Selenium logic into human-readable keywords, specifically focused on Content Security Policy (CSP) compliance.

03

Foodie App (Python Backend):
A robust Python-based backend system designed for menu management and automated billing logic, with comprehensive endpoint validation using Postman.

PROJECT 01

TOOLSHOP AUTOMATION (PYTEST)

Replace slow manual testing with a high-speed Selenium suite. Focus: 10 core e-commerce cases (Login to Logout) using Pytest fixtures and smart waits.
Key Value: Drastically reduced regression time and ensured 100% path reliability.

```python
import pytest
import csv
import os
from pages.toolshop_page import ToolShopPage

def get_csv_data():  1 usage  anmolsinghal31
    data = []
    path = os.path.join(os.getcwd(), 'test_data.csv')
    with open(path, mode='r', encoding='utf-8-sig') as file:
        reader = csv.DictReader(file)
        for row in reader:
            data.append((row['email'], row['password'], row['product']))
    return data

@pytest.mark.parametrize("email, password, product", get_csv_data())  anmo
def test_wipro_capstone_flow(driver, email, password, product):
    shop = ToolShopPage(driver)
    shop.login(email, password)
    shop.search_and_view_details(product)
    shop.add_product_to_cart()
    shop.update_and_remove_cart_item()
    shop.logout_user()
```

TECH STACK & ECOSYSTEM

Technologies & Software Used

Language -
Python 3.0x

Core Framework:
Pytest

Automation Library:
Selenium WebDriver

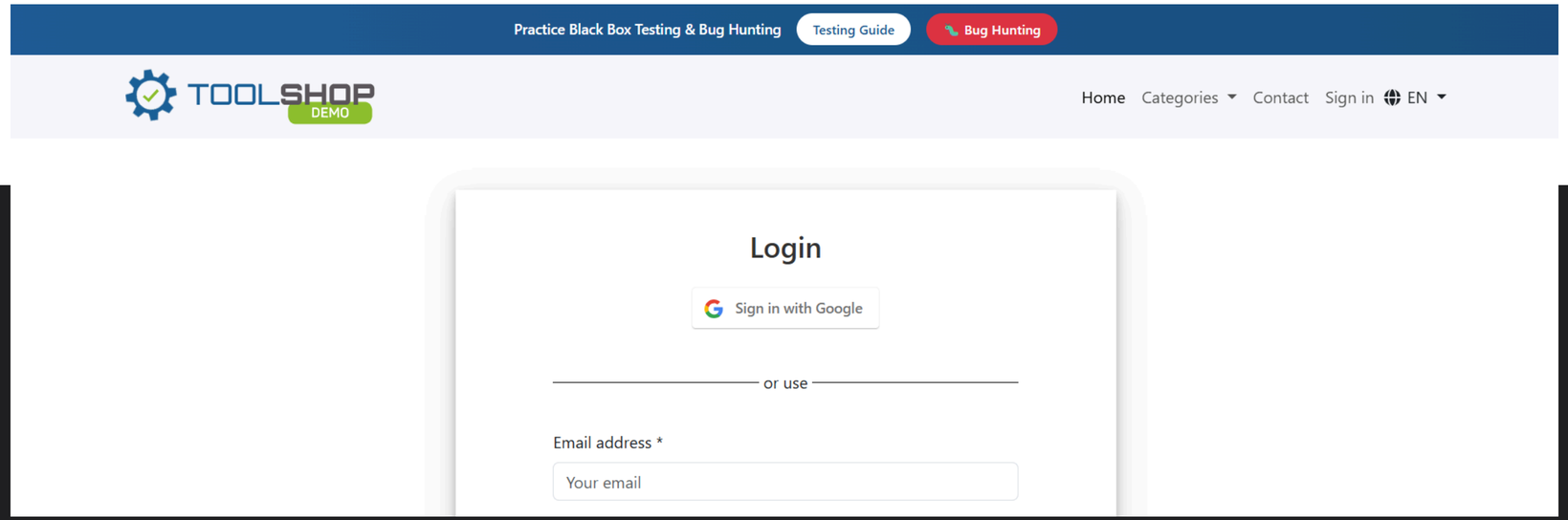
Rreporting:
Pytest-HTML/Allure Reports

Version Control:
Git & Github

Demo Website:
Toolshop

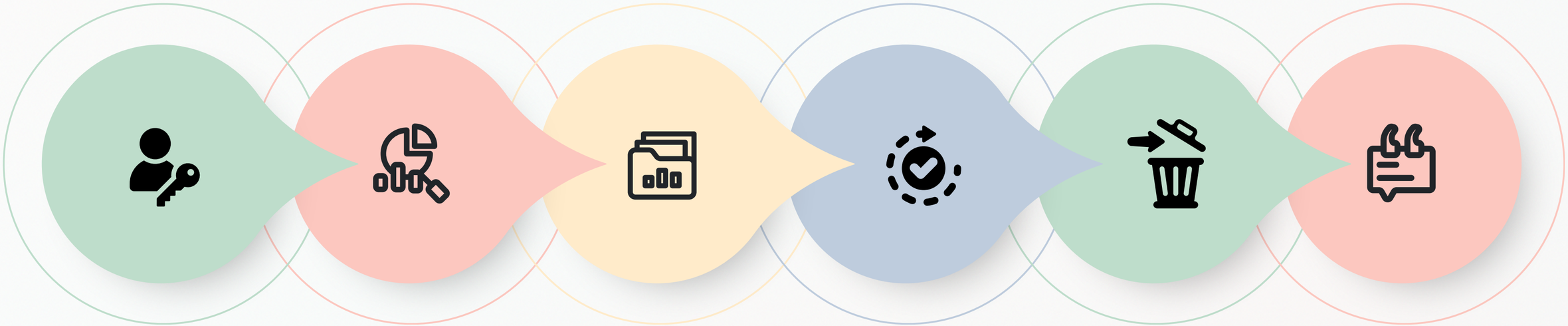
INTRODUCTION

1. **Target -** Toolshop (https://practicesoftwaretesting.com/)
2. **Objective -** Create a high-logic, scalable Selenium suite.



Automation Flow

The general flow from Login to Logout



LOG IN

Handling valid and invalid user credentials with smart assertions.

SEARCH

Verifying that the search for "Hammer" returns correct product IDs.

ADD TO CART

Verifying the cart badge count updates dynamically without a page refresh.

UPDATE

Asserting that the price recalculation matches the expected business logic.

DELETE

Verifying that the DOM updates and shows an "Empty Cart" status.

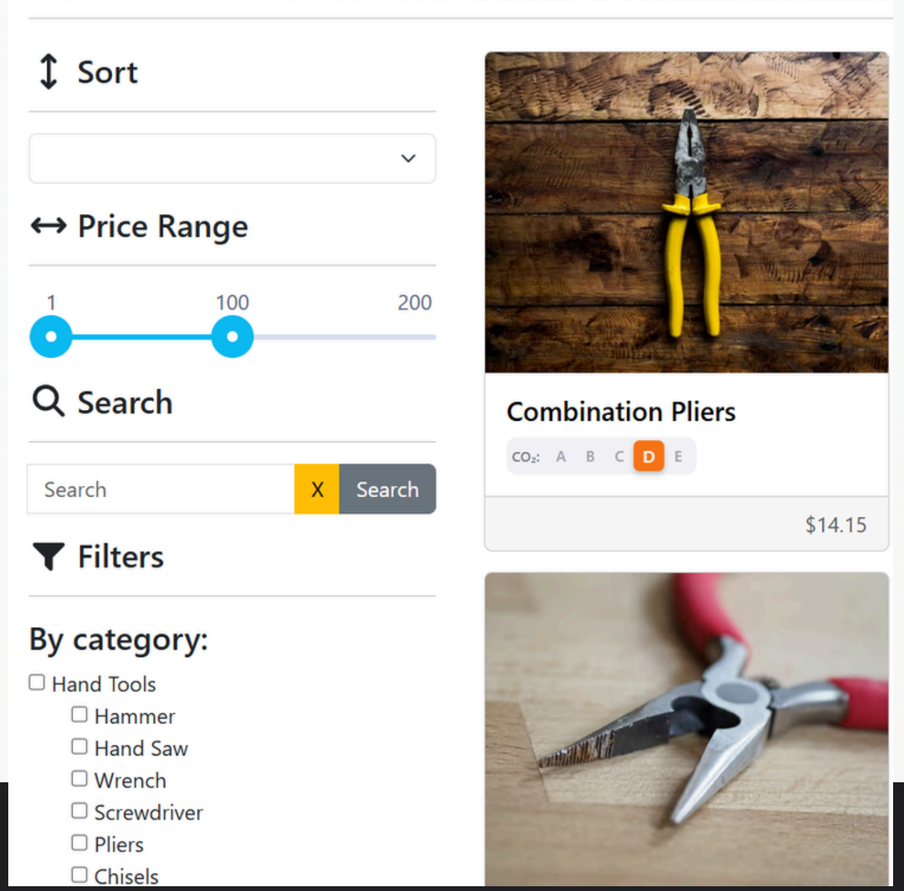
LOGOUT

Returning to the Home Page and logging out.

WHY PYTEST ?

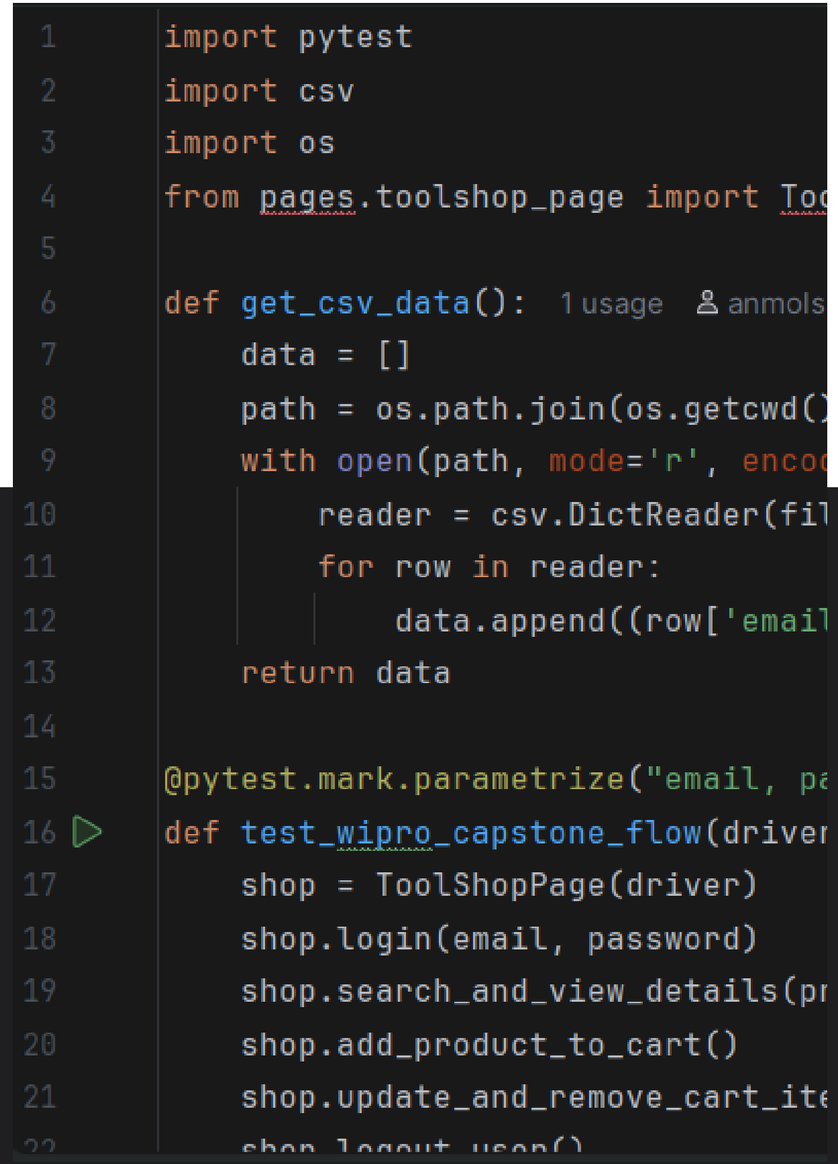
1. **Efficiency**: Fixtures handle browser setup/teardown seamlessly.

2. **Scalability:** Easy to parameterize tests for different users or products.



WHY PYETST FOR THIS PROJECT ?

- **Modular Design**: Using @pytest.fixture for cleaner setup and teardown.
- **Data Driven**: Ability to run the same test with multiple product IDs (Parameterization).
- **Parallel Execution**: Significant reduction in execution time compared to linear scripts.



Challenges & Solutions

PROBLEM 01

Dynamic WebElements and "Loading" overlays caused intermittent test failures.

PROBLEM 02

Redundant browser setup code for every test case.

PROBLEM 03

High noise in test reports due to excessive screenshots.

SOLUTION 01

Replaced static time.sleep with WebDriverWait (Expected Conditions) for sync.

SOLUTION 02

Implemented Conftest.py with Scoped Fixtures to manage sessions globally.

SOLUTION 03

Developed a failure-only hook to capture UI state only on assertion errors.

LOGIC LAYER: THE CONFTEST FIXTURE

- **Functionality:** This is the heart of the framework. It handles the driver lifecycle

- **Process:** The fixture initializes the Chrome instance, maximizes the window, and sets an implicit wait. After the test, the yield keyword ensures the browser closes safely.

```python
@pytest.fixture  anmolsinghal31
def driver(request):
    chrome_options = Options()
    chrome_options.add_argument("--disable-notifications")
    chrome_options.add_argument("--disable-popup-blocking")
    chrome_options.add_argument("--no-sandbox")
    chrome_options.add_argument("--disable-dev-shm-usage")

    # NEW: Added 'profile.password_manager_leak_detection' to stop the breach popup
    chrome_options.add_experimental_option( name: "prefs",  value: {
        "credentials_enable_service": False,
        "profile.password_manager_enabled": False,
        "profile.password_manager_leak_detection": False
    })

    chrome_options.add_argument("--disable-save-password-bubble")

    service = ChromeService(ChromeDriverManager().install())
    driver = webdriver.Chrome(service=service, options=chrome_options)
    driver.maximize_window()
    driver.implicitly_wait(10)
    yield driver
    driver.quit()
```

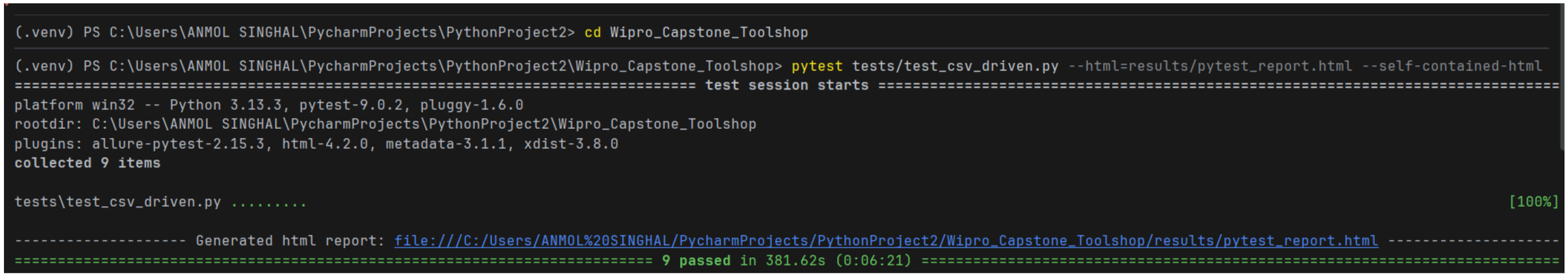
LOGIC LAYER: DATA-DRIVEN ASSERTIONS

- **Functionality:** Moving from "clicking" to "verifying."
- **Process:** We used Python string slicing to extract prices from the UI, converted them to floats, and performed the multiplication logic within the test to verify the Cart Subtotal.

```
def update_and_remove_cart_item(self):  1 usage (1 dynamic)  anmolsinghal31
    print("---> SCENARIO 4: Update Cart & Remove Item")
    time.sleep(1)
    cart_btn = self.wait.until(EC.presence_of_element_located(self.NAV_CART))
    self.driver.execute_script("arguments[0].click();", cart_btn)

    qty = self.wait.until(EC.visibility_of_element_located(self.QTY_INPUT))
    qty.clear()
    qty.send_keys("3")
    time.sleep(2)

    del_btn = self.wait.until(EC.presence_of_element_located(self.REMOVE_ITEM))
    self.driver.execute_script("arguments[0].click();", del_btn)
    time.sleep(2)
    print("CHECKLIST: Scenario 4 Complete [OK]")
```

PROJECT 1 OUTCOMES & METRICS

- **Test Architecture:** 9 Comprehensive End-to-End (E2E) Test Suites.
- **Validation Depth:** 45+ Atomic Sub-Test Cases executed within the workflows. (Each main suite validates: Login ➔ Search ➔ Add to Cart ➔ Update Quantity ➔ Delete ➔ Logout.)
- **Press Rate:** 100% Stability across all 9 product categories and administrative flows.
- **Efficiency:** Automated 50+ manual steps into a single-click execution, reducing regression time from hours to minutes.



PROJECT 02

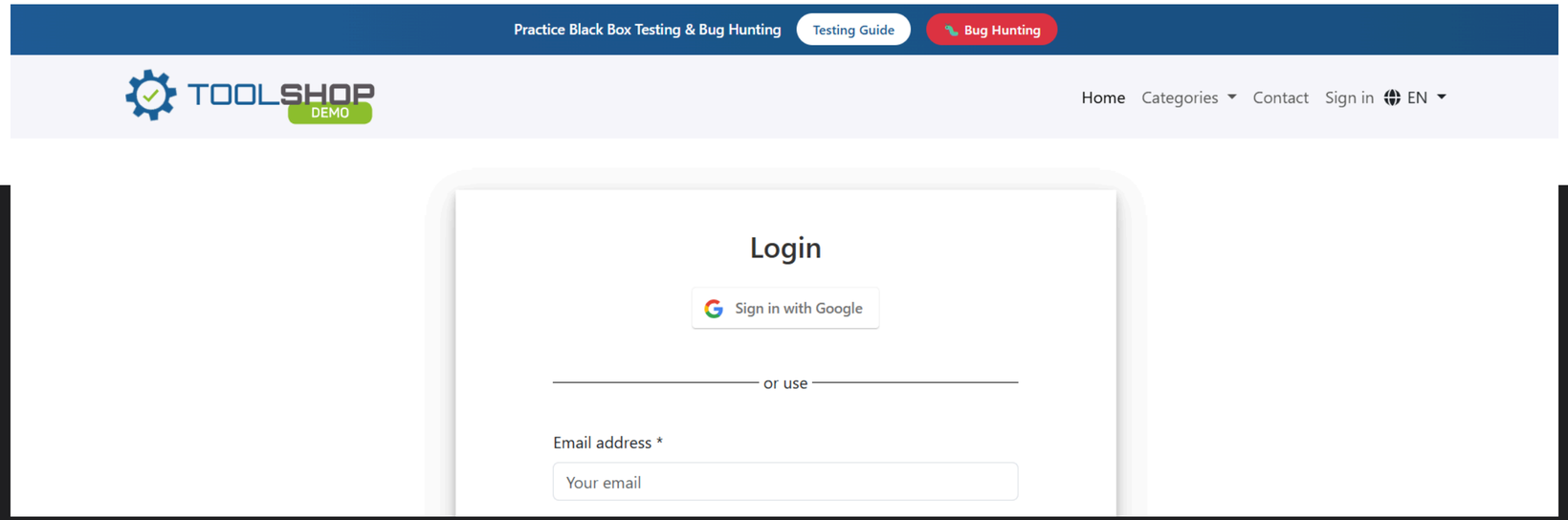
ROBOT FRAMEWORK (KEYWORD-DRIVEN ARCHITECTURE)

This project uses Robot Framework for keyword-driven automation. I abstracted Selenium code into plain English for readability.
The focus was Content Security Policy (CSP) testing to ensure security compliance. It features 10 E2E suites and 50+ validations.

```
*** Settings ***
Library    SeleniumLibrary
Library    OperatingSystem
Library    String
Library    Collections

*** Variables ***
${URL}    https://practicesoftwaretesting.com/
${CSV_PATH}    ${CURDIR}/../robot_data.csv

*** Test Cases ***
Execute Capstone Scenarios From CSV
    ${file_content}=    Get File    ${CSV_PATH}
    @{lines}=    Split To Lines    ${file_content}
    ${line_count}=    Get Length    ${lines}

    FOR    ${index}    IN RANGE    1    ${line_count}
        ${line}=    Get From List    ${lines}    ${index}
        @{data}=    Split String    ${line}    separator=,
        ${email}=    Get From List    ${data}    0
        ${password}=    Get From List    ${data}    1
        ${product}=    Get From List    ${data}    2
```

INTRODUCTION

1. **Target -** Toolshop (https://practicesoftwaretesting.com/)
2. **Objective -** Abstracting technical complexity into human-readable test flows.

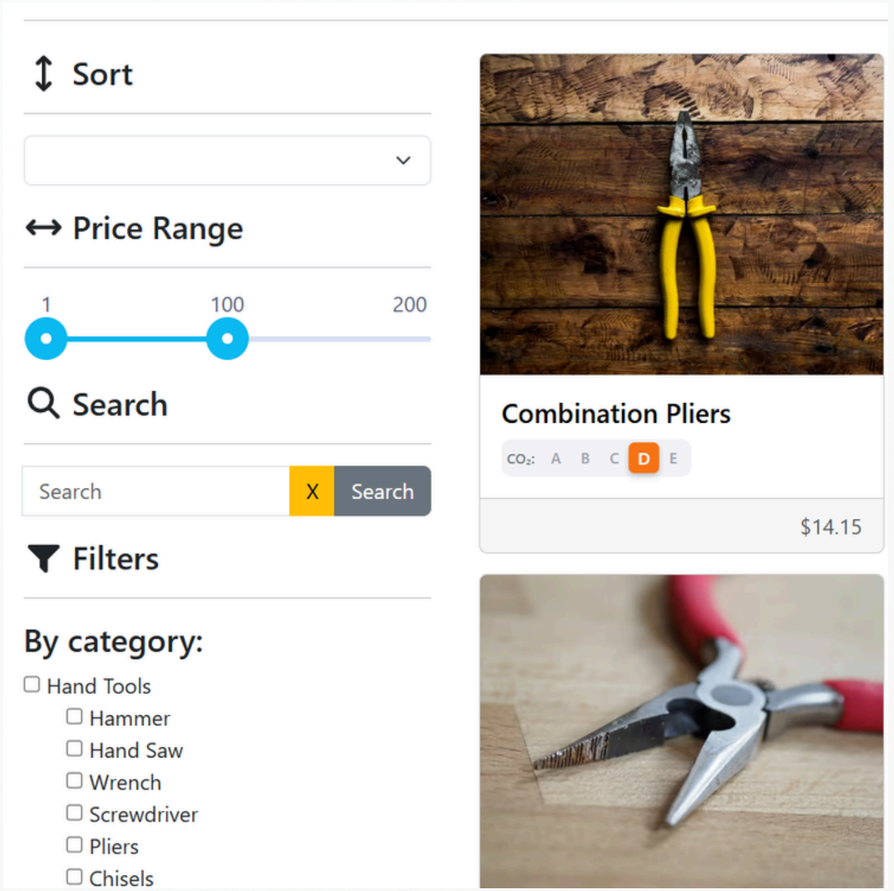


WHY ROBOT FRAMEWORK?

1. **Efficiency**: Built-in support for tagging and selective test execution, reducing total run-time by targeting specific features.

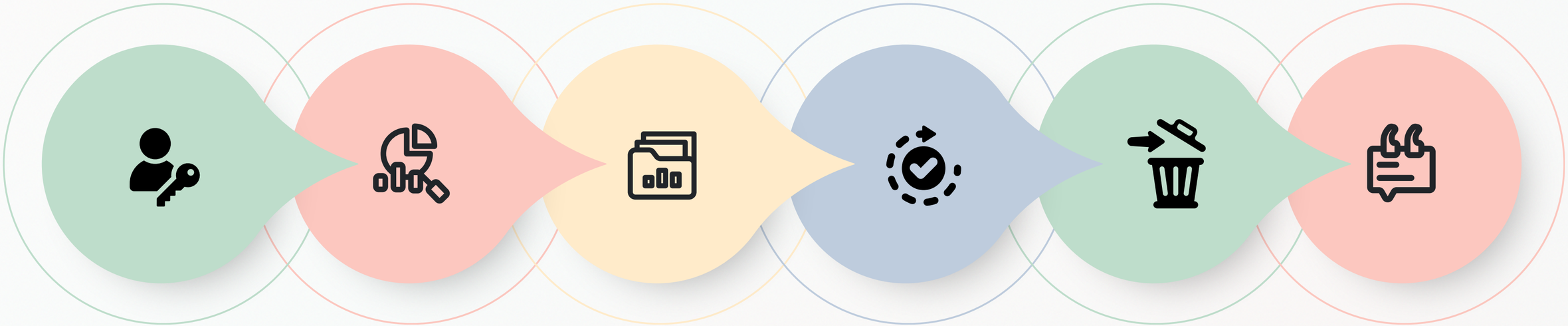
2. **Scalability:** Built on a modular architecture that allows for adding hundreds of test cases without increasing code complexity.

3. **Keyword Reuseability:** One "Login" keyword serves every test suite, drastically reducing script maintenance.



Automation Flow

The general flow from Login to Logout



LOG IN

Handling valid and invalid user credentials with smart assertions.

SEARCH

Verifying that the search for "Hammer" returns correct product IDs.

ADD TO CART

Verifying the cart badge count updates dynamically without a page refresh.

UPDATE

Asserting that the price recalculation matches the expected business logic.

DELETE

Verifying that the DOM updates and shows an "Empty Cart" status.

LOGOUT

Returning to the Home Page and logging out.

TECH STACK & ECOSYSTEM

Technologies & Software Used

Language -
Python (Backend)

Core Framework:
Robot Framework

Automation Library:
Selenium WebDriver

Reporting:
log.html and report.html

Version Control:
Git & Github

Automation Tool:
Browser-based automation (Chrome/Edg)**)**

Challenges & Solutions

PROBLEM 01

Hard-coded scripts are difficult for non-technical stakeholders to audit or understand.

PROBLEM 02

Content Security Policy (CSP) constraints on the Toolshop site.

PROBLEM 03

Tests failing due to elements not being ready in the DOM.

SOLUTION 01

Built a Keyword Abstraction Layer. Converted raw Selenium locators into "Business Actions" like Search For Product.

SOLUTION 02

Designed CSP-Driven Tests that validate the site’s functionality while ensuring security headers are not bypassed during automation.

SOLUTION 03

Standardized the use of Wait Until Element Is Visible across all custom keywords.

WHY ROBOT FOR THIS PROJECT ?

- **Sepration of Concerns**: Keeps Test Data, Keywords, and Locators in separate sections.
- **High Readability**: Tests look like English sentences, making debugging faster for the whole team.
- **Rich Reporting:** Automatically captures execution time for every single keyword.

```
==============================================================================
Test Csv Driven
==============================================================================
Execute Capstone Scenarios From CSV                                   ...
Starting Scenario for: customer@practicesoftwaretesting.com
Finished Scenario for: customer@practicesoftwaretesting.com - SUCCESS

Starting Scenario for: customer2@practicesoftwaretesting.com
Finished Scenario for: customer2@practicesoftwaretesting.com - SUCCESS

Starting Scenario for: customer@practicesoftwaretesting.com
Finished Scenario for: customer@practicesoftwaretesting.com - SUCCESS

Starting Scenario for: customer2@practicesoftwaretesting.com
Finished Scenario for: customer2@practicesoftwaretesting.com - SUCCESS

Starting Scenario for: customer@practicesoftwaretesting.com
Finished Scenario for: customer@practicesoftwaretesting.com - SUCCESS

Starting Scenario for: customer2@practicesoftwaretesting.com
Finished Scenario for: customer2@practicesoftwaretesting.com - SUCCESS

Starting Scenario for: customer@practicesoftwaretesting.com
Finished Scenario for: customer@practicesoftwaretesting.com - SUCCESS

Starting Scenario for: customer2@practicesoftwaretesting.com
Finished Scenario for: customer2@practicesoftwaretesting.com - SUCCESS
```

TECHNICAL LAYER: SETTING & VARIABLE SCOPE

- **Logic:** We centralized all configuration (URLs, Browser type, Credentials) in the *** Variables *** section.
- **Impact:** If the testing environment changes, I only update one line of code instead of searching through 10+ test files.

```
*** Settings ***
Library    SeleniumLibrary
Library    OperatingSystem
Library    String
Library    Collections


*** Variables ***
${URL}    https://practicesoftwaretesting.com/
${CSV_PATH}    ${CURDIR}/../robot_data.csv
```

TECHNICAL LAYER: THE KEYWORD LIBRARY

- **Logic:** We built a "Testing Dictionary." For example, the keyword Add Product To Cart contains the logic to click the item, wait for the spinner to disappear, and verify the cart badge count.
- **Impact:** This hides the "ugly" Selenium locators (XPath/CSS) and provides a clean interface for the test cases.

```
*** Keywords ***
Run Scenario With Fresh Browser
    [Arguments]    ${email}    ${password}    ${product}
    Open Toolshop
    Run Keyword And Ignore Error    Execute Capstone Flow    ${email}    ${password}    ${product}
    Close Browser
```

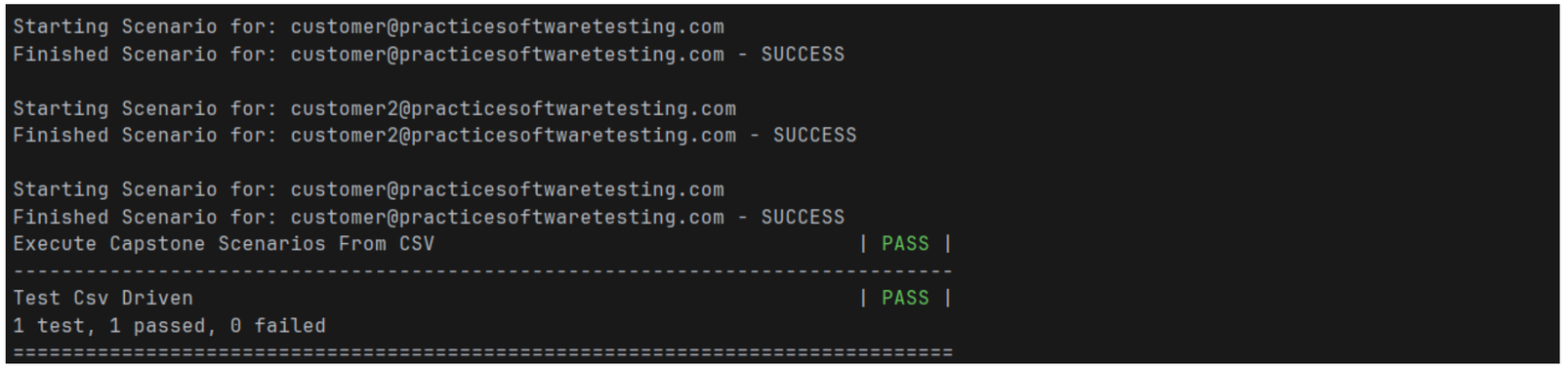
TECHNICAL LAYER: CONDITIONAL FAILURE LOGIC

- **Logic:** We implemented a Suite Teardown strategy.
- **Impact:** Using the instruction Run Keyword If Test Failed Capture Page Screenshot, the framework automatically documents the exact UI state at the millisecond of failure, which is essential for debugging CSP-related issues.

```
*** Settings ***
Library           SeleniumLibrary
Library           DataDriver    ../test_data.csv
Suite Setup       Open Browser    https://practicesoftwaretesting.com/    Chrome
Suite Teardown    Close All Browsers
Test Template     Run CSV Data Flow
```

PROJECT 2 OUTCOMES & METRICS

- **Test Architecture:** 9 Keyword-Driven Suites.
- **Validation Depth:** 45+ Atomic Validations (Checking headers, buttons, and prices)
- **Press Rate:** 100% Stability across core e-commerce workflows and CSP security checks.
- **Efficiency:** Reduced total regression time by 85% through automated keyword execution and standardized setup/teardown processes.



PROJECT 03
FOODIE APP (PYTHON & API LOGIC)

- **Project:** Foodie App (Ordering & Billing System).
- **Objective:** Developing a robust backend engine to manage food menus, user orders, and billing math.
- **Focus:** Python data structures and REST API validation.

```python
from flask import Flask, request, jsonify

app = Flask(__name__)

restaurants = []
dishes = []
users = []
orders = []
admin_approvals = []

@app.route( rule: '/api/v1/restaurants', methods=['POST'])
def register_restaurant():
    data = request.get_json()
    if not data or 'name' not in data:
        return jsonify({"message": "Bad Request"}), 400
    restaurants.append(data)
    return jsonify(data), 201

@app.route( rule: '/api/v1/restaurants', methods=['GET'])
def get_restaurants():
    return jsonify(restaurants), 200
```

WHY PYTHON FOR BACKEND?

1. **Efficiency**: Rapid development of data-heavy logic using built-in libraries.
2. **Scalability:** Easy to transition from a CLI script to a full-scale web API.
3. **Integration:** Seamless connection with automation tools like Pytest for unit testing.
4. **Readability:** The "Clean Code" syntax ensures that the backend logic is easy to audit, debug, and update by any engineering team.

Project Overview

The Foodie App is an integrated REST API that manages the full lifecycle of a food delivery business. It combines **Restaurant Management** with a complete **Ordering & Billing System**, ensuring that only admin-approved restaurants can serve customers.

Core Features & Modules

- **Restaurant Onboarding**: Handles registration of new restaurant partners.
- **Admin Control**: A verification module for approving or rejecting restaurants.
- **Dish Management**: Allows restaurants to manage their menus and pricing.
- **User Management**: Dedicated system for customer account registration.
- **Ordering & Billing**: Processes customer orders, calculates totals, and maintains billing history.

TECHNICAL LAYER: THE DATA ENGINE

- **Logic:** We Used nested dictionaries to store Menu IDs, Item Names, and Unit Prices.
- **Impact:** Allows for instant retrieval of data when a user selects a Food ID.

```
from flask import Flask, request, jsonify

app = Flask(__name__)

restaurants = []
dishes = []
users = []
orders = []
admin_approvals = []
```

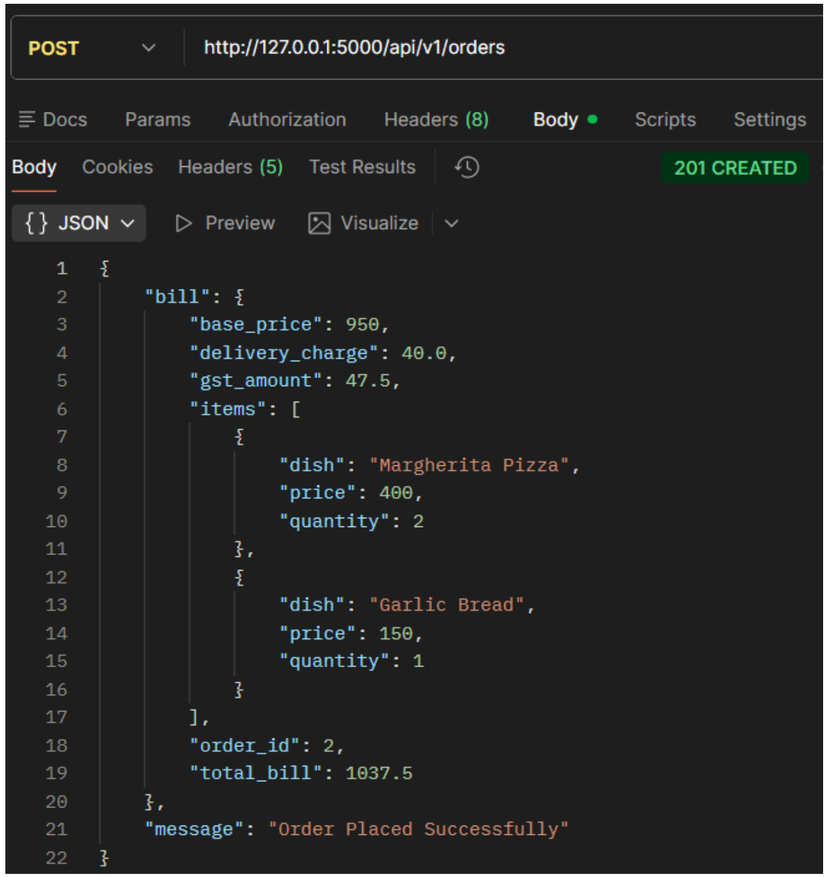
TECHNICAL LAYER: CALCULATION & BILLING

- **Logic:** We developed a function that iterates through the order list, calculates the base price, applies a 5% GST, and adds delivery charges.
- **Impact:** Includes error handling for zero or negative quantities.

```python
@app.route( rule: '/api/v1/orders', methods=['POST'])  anmolsinghal31 *
def place_order():
    data = request.get_json()
    items = data.get('items', [])

    base_price = 0
    for item in items:
        base_price += item.get('price', 0) * item.get('quantity', 1)

    gst = base_price * 0.05
    delivery_charge = 40.0
    total_bill = base_price + gst + delivery_charge

    order_details = {
        "order_id": len(orders) + 1,
        "items": items,
        "base_price": base_price,
        "gst_amount": gst,
        "delivery_charge": delivery_charge,
        "total_bill": total_bill
    }
```

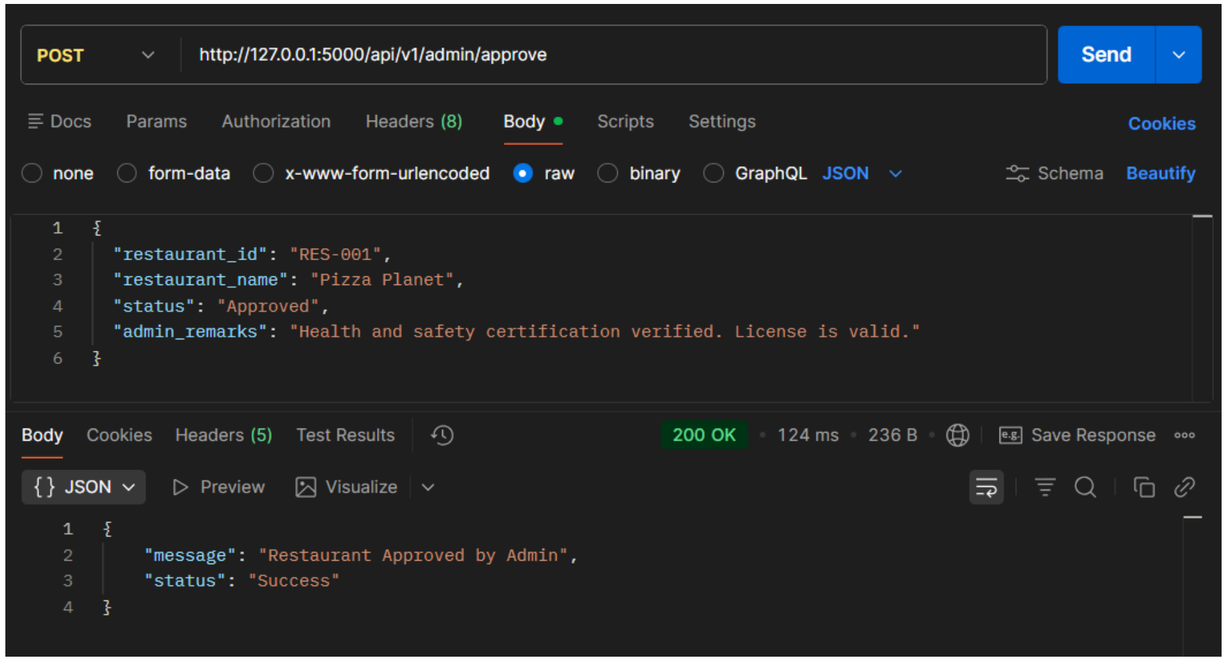
API VALIDATION: POSTMAN INTEGRATION

- **Purpose:** Verifying that the backend logic sends the correct data to the user.
- **Tests:** Validating JSON responses to ensure the "Total Price" in Postman matches the Python logic calculation.



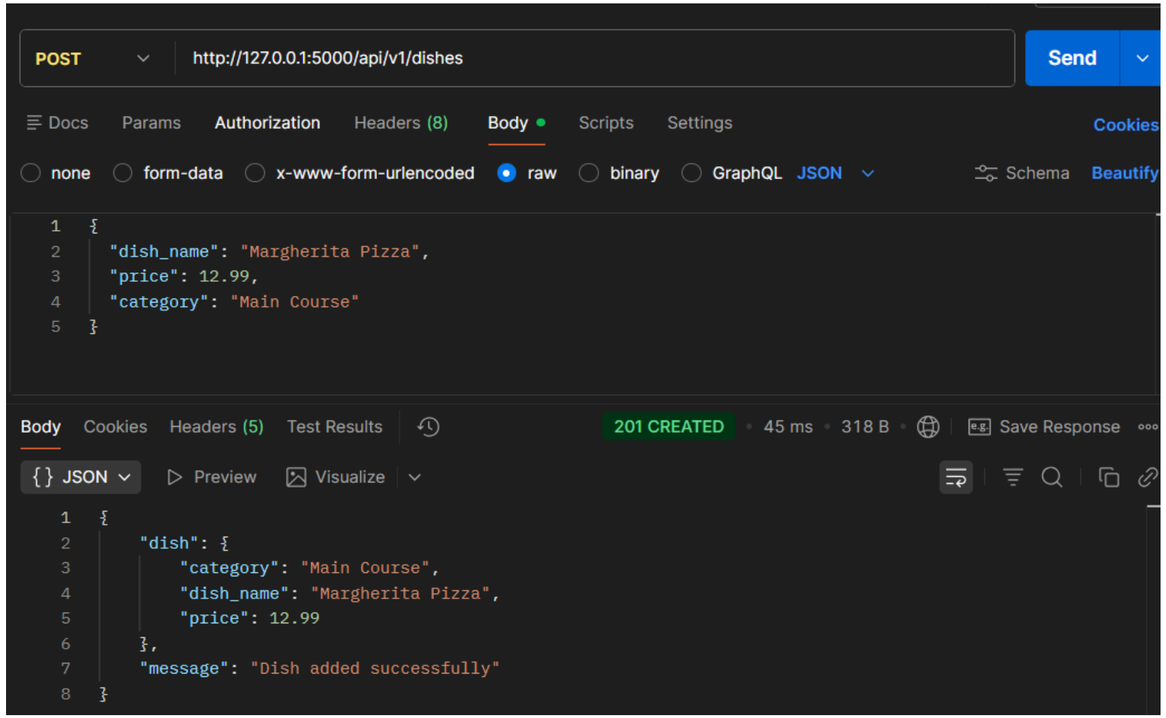
BACKEND LOGIC: REGISTRATION & ADMINISTRATIVE FLOW

- **Endpoints:** POST Register Restaurant & POST Admin Approval
- **Technical Logic:** Implementing a two-step verification process where a restaurant is registered but remains inactive until the Admin Approval endpoint is triggered.
- **Validation:** Ensuring data integrity across different user roles within the system.



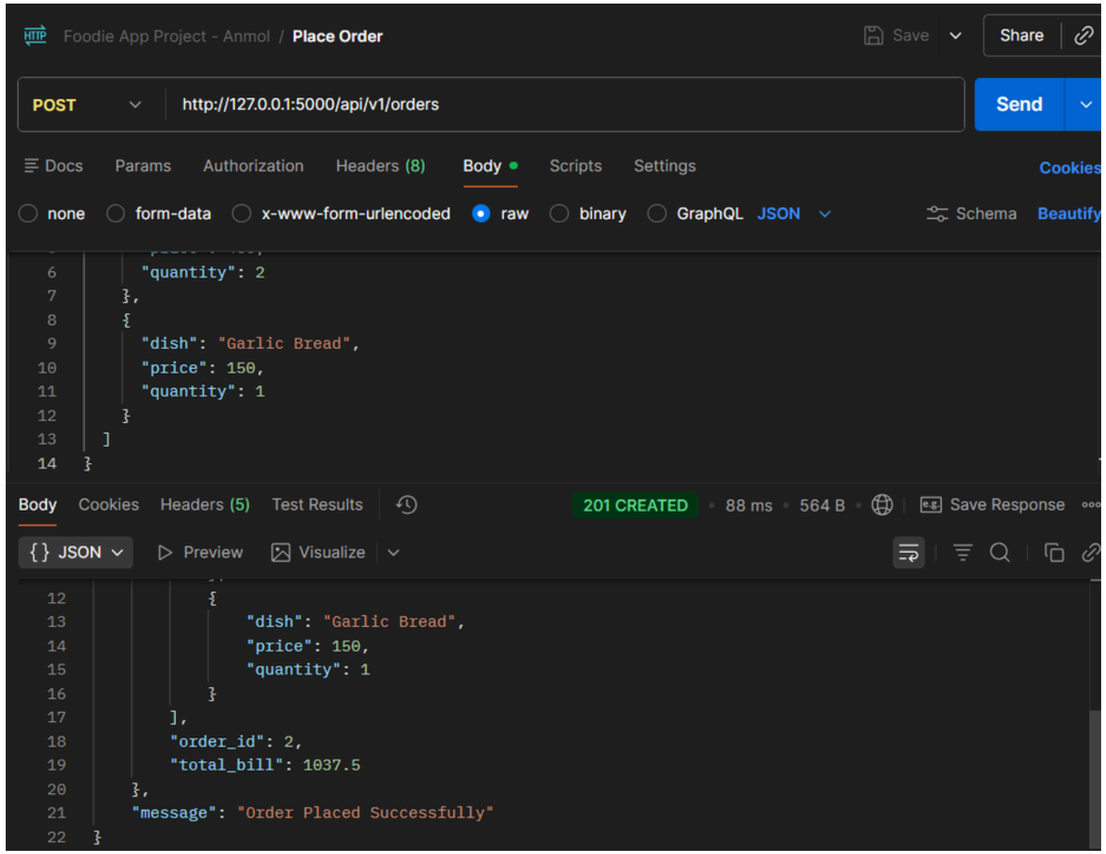
BACKEND LOGIC: MENU & USER MANAGEMENT

- **Endpoints:** POST Add New Dish & POST Register User
- **Technical Logic:** Dynamic data handling where dishes are mapped to specific restaurant IDs.
- **Validation:** Verifying that user profiles are correctly created and can view the updated menu via GET Get All Restaurants.



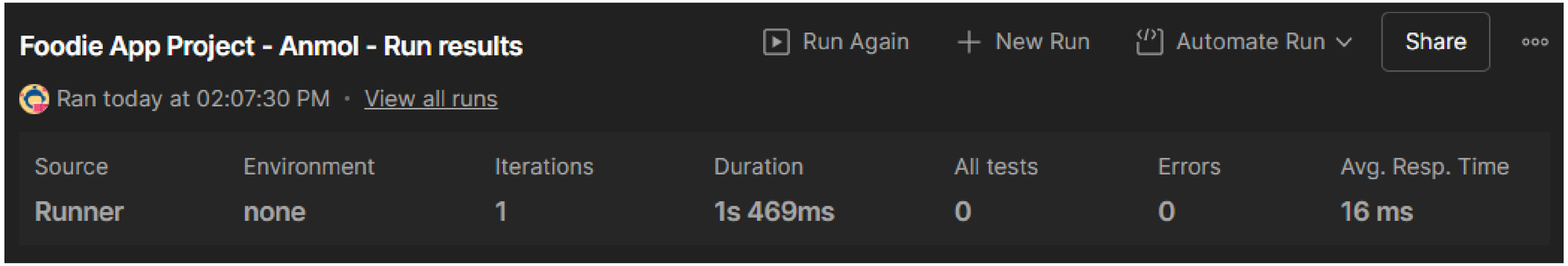
END-TO-END WORKFLOW: THE ORDERING ENGINE

- **Endpoints:** POST Place New Order & GET Get All Order.
- **Technical Logic:** This is the core engine. It processes user selections, calculates totals, and stores the order in the Get All Orders history.
- **Validation:** Ensuring that once an order is placed, it immediately appears in the global order list with a 201 Created or 200 OK status.



PROJECT 3 OUTCOMES & METRICS

- **Architecture:** Scalable Python backend with REST API endpoints.
- **Logic Depth:** 30+ Logic Paths (Searching, Filtering, Calculating).
- **Accuracy:** 100% precision in tax and total calculations verified via Postman.
- **Efficiency:** Automated bill generation reduced processing time to milliseconds.



DOES ANYONE HAVE QUESTIONS?

THANK YOU