1. SOFTWARE DEVELOPMENT LIFECYCLE (SDLC) FOR AN E-COMMERCE PLATFORM

AIM:

To understand and implement the Software Development Lifecycle (SDLC) for an e-commerce platform by following various phases including Requirement Analysis, Planning, Design, Development, Testing, and Deployment.

SOFTWARE & HARDWARE REQUIREMENTS:

Software:

- React.js
- Node.js
- Spring Boot
- MySQL
- Postman
- Docker
- AWS
- JIRA
- Trello

PROCEDURE:

STEP 1: Requirement Analysis

1. Conduct stakeholder meetings (business owners, customers, technical teams).
2. Identify key requirements such as:
    - User Management: Signup, login, user profile.
    - Product Listing: Categories, filters, search functionality.
    - Shopping Cart & Checkout: Add to cart, apply discounts, calculate taxes.
    - Payment Gateway: Integrate Razorpay, Stripe, or PayPal.
    - Order Management: Order placement, status tracking.
    - Admin Panel: Manage products, orders, and users.
3. Document findings in a Software Requirement Specification (SRS).
4. Get stakeholder approval before proceeding to planning.

STEP 2: Planning

1. Divide the project into 6 Agile sprints (each sprint = 4 weeks).
2. Assign tasks to respective teams (Frontend, Backend, Database, QA).
3. Use JIRA or Trello for task management.
4. Define project risks (e.g., API integration delays, third-party payment issues).
5. Create a Gantt chart for progress tracking.

Example Sprint Breakdown:

Sprint	Task
1	Set up project, UI wireframes, database schema
2	User authentication (signup/login)
3	Product listing & search features
4	Shopping cart & checkout integration
5	Payment gateway integration & order tracking
6	Testing, bug fixes, and deployment

STEP 3: Design

1. **Database Schema (MySQL):** Define tables (users, products, orders, cart, payments).
2. **Frontend (React):** Design UI using Figma or Adobe XD.
3. **Backend (Node.js & Spring Boot):** Define REST APIs for user login, product retrieval, order management.
4. **System Architecture:**
    - **Frontend:** React UI → API calls
    - **Backend:** Node.js (Product Management) + Spring Boot (Order Management)
    - **Database:** MySQL (Stores products, orders, users)
    - **Payment:** Razorpay API

STEP 4: Development

1. **Setup GitHub Repository:** Use branching strategy (main, dev, feature-*).
2. **Frontend Development:**
    - Build components (ProductCard, CartItem, CheckoutForm), fetch API data using Axios.
3. **Backend Development:**
    - Implement JWT-based authentication (Spring Security), create REST APIs.
4. **Payment Gateway Integration:**
    - Configure Razorpay/Stripe API, implement webhook for payment success/failure.
5. **Testing APIs using Postman:** Verify authentication, order placement, payment responses.
6. **Error Handling & Logging:** Use try-catch blocks, integrate logging with Log4j.

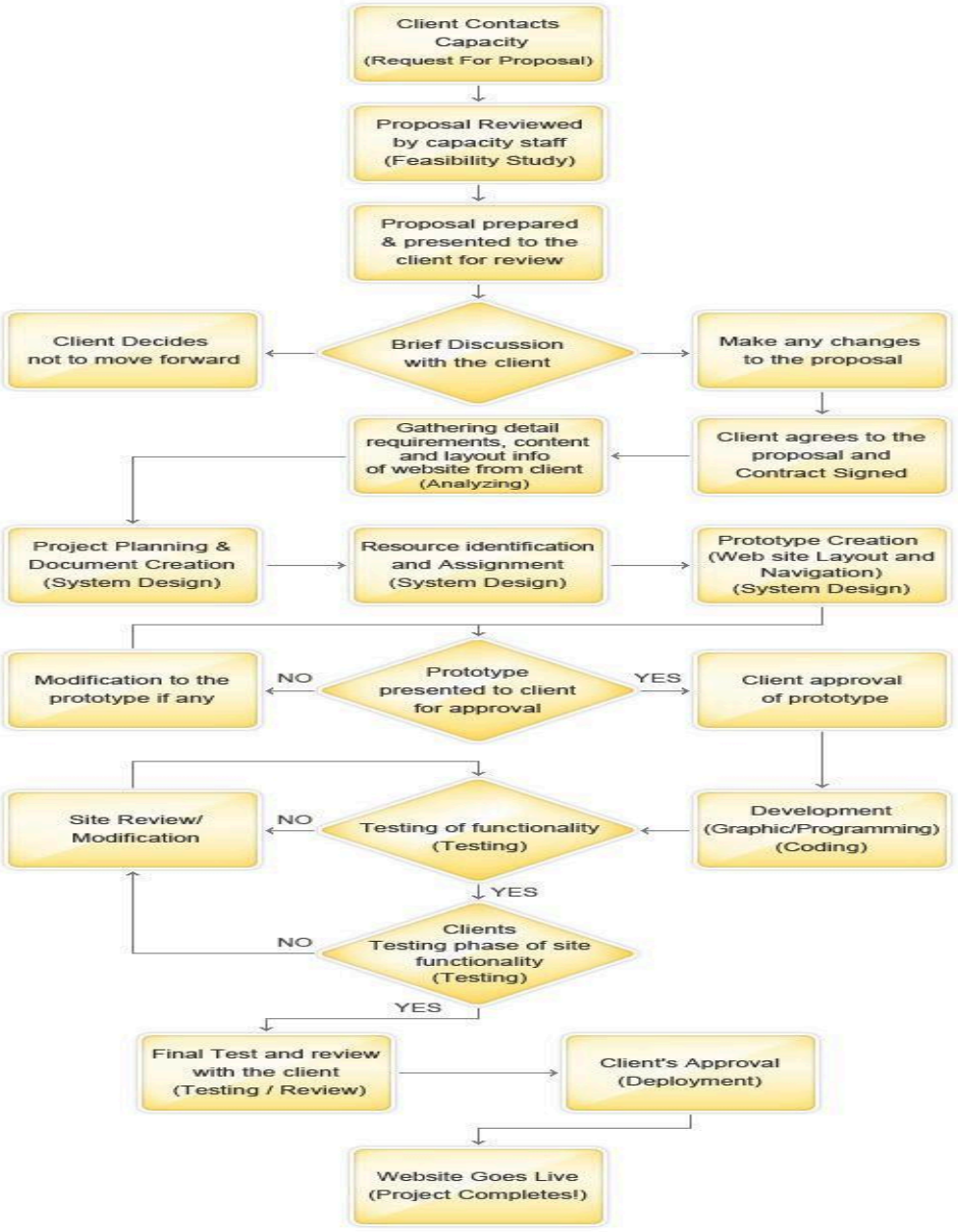
STEP 5: Testing

1. **Unit Testing:** JUnit (Java), Jest (React) for functionality validation.
2. **API Testing:** Postman & Newman to validate endpoints.
3. **UI Testing:** Automate login, cart operations using Selenium WebDriver.
4. **Security Testing:** Prevent SQL Injection, encrypt passwords with BCrypt.
5. **Performance Testing:** Use JMeter to simulate high traffic.

STEP 6: Deployment

1. **Dockerize Application:** Create Dockerfile for frontend, backend, use docker-compose.
2. **AWS Deployment:**
    - Setup EC2 instance, install dependencies.
    - Run docker pull and start containers.
3. **Database Hosting:** Use Amazon RDS (MySQL) for production.
4. **CI/CD Integration:** Automate deployment with GitHub Actions & Jenkins.
5. **Monitoring:** Use Prometheus + Grafana for real-time logs.
6. **Final Testing:** Validate live transactions, user logins, and order placements.

OUTPUT:



RESULT:

Thus, the Software Development Lifecycle (SDLC) for an e-commerce platform was successfully implemented and tested.

2. Building a Quality Assurance Plan for a sample project.

Aim:

To build a Quality Assurance Plan for a sample project using JIRA to track and manage testing activities, defects, and project quality.

Procedure:

Step 1: Define Quality Assurance Objectives

- Ensure system reliability, functionality, and performance meet the project requirements.
- Identify and track defects efficiently to improve software quality.
- Implement structured test case management and execution.

Step 2: Set Up a JIRA Project for QA

1. Sign Up & Create a New JIRA Project
   - Go to JIRA Cloud (https://www.atlassian.com/software/jira).
   - Create a new project and select Scrum/Kanban template.
1. Define Issue Types (Customize JIRA for QA)
   - Bug – Tracks software defects.
   - Test Case – Documents test scenarios.
   - Task – Assigns QA-related work.

Step 3: Define the Testing Process in JIRA

Phase	Action in JIRA
Test Planning	Create test cases & link to user stories.
Test Execution	Run test cases & log defects if any fail.
Bug Tracking	Report, assign, and track defects using workflows.
Reporting	Generate test coverage & defect density reports.

Step 4: Create & Manage Test Cases in JIRA

1. Go to JIRA → Test Cases
2. Click "Create Issue" → Select "Test Case"
3. Define:
   - Title: "Login Page - Valid Credentials Test"
   - Preconditions: User must have a valid account
   - Steps: Enter username → Enter password → Click login
   - Expected Result: User should be redirected to the dashboard
1. Link Test Case to a User Story
   - Connect test cases to project tasks for traceability.

Step 5: Log and Track Defects

1. If a test case fails, click "Create Bug" in JIRA.

2. Fill in:
   - Title: "Login Button Not Working"
   - Description: Clicking "Login" does not redirect the user.
   - Priority: High
   - Status: Open
   - Assignee: Developer responsible for fixing the issue.
1. Monitor Defect Lifecycle
   - Open → In Progress → Resolved → Closed

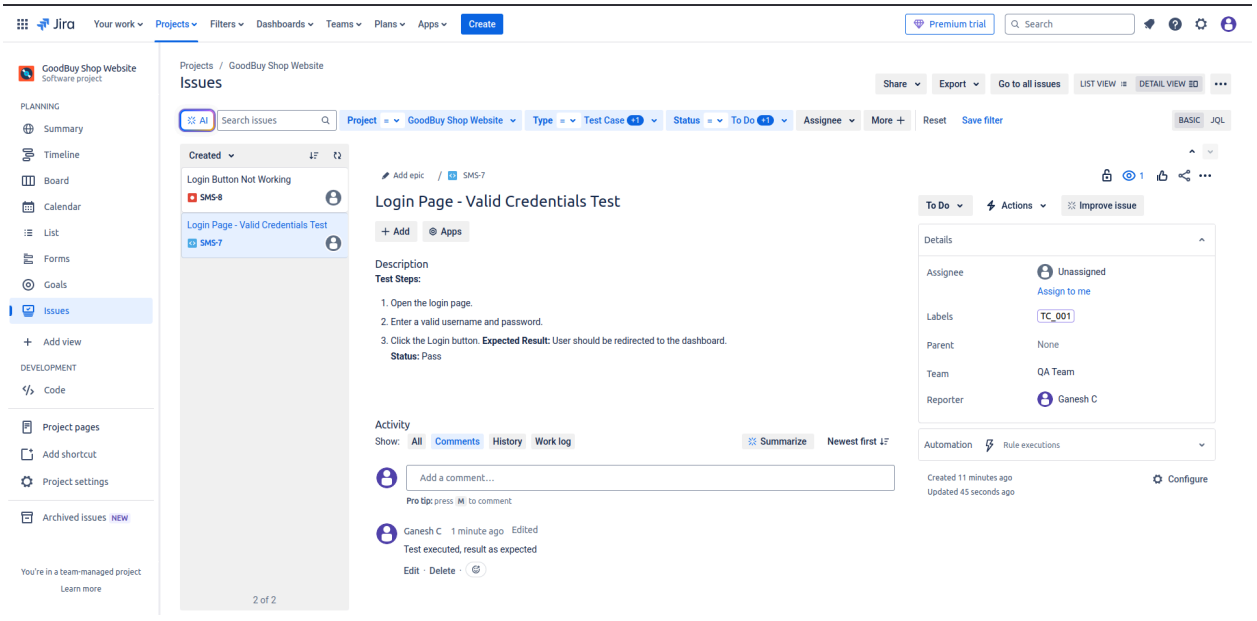
Step 6: Generate QA Reports in JIRA

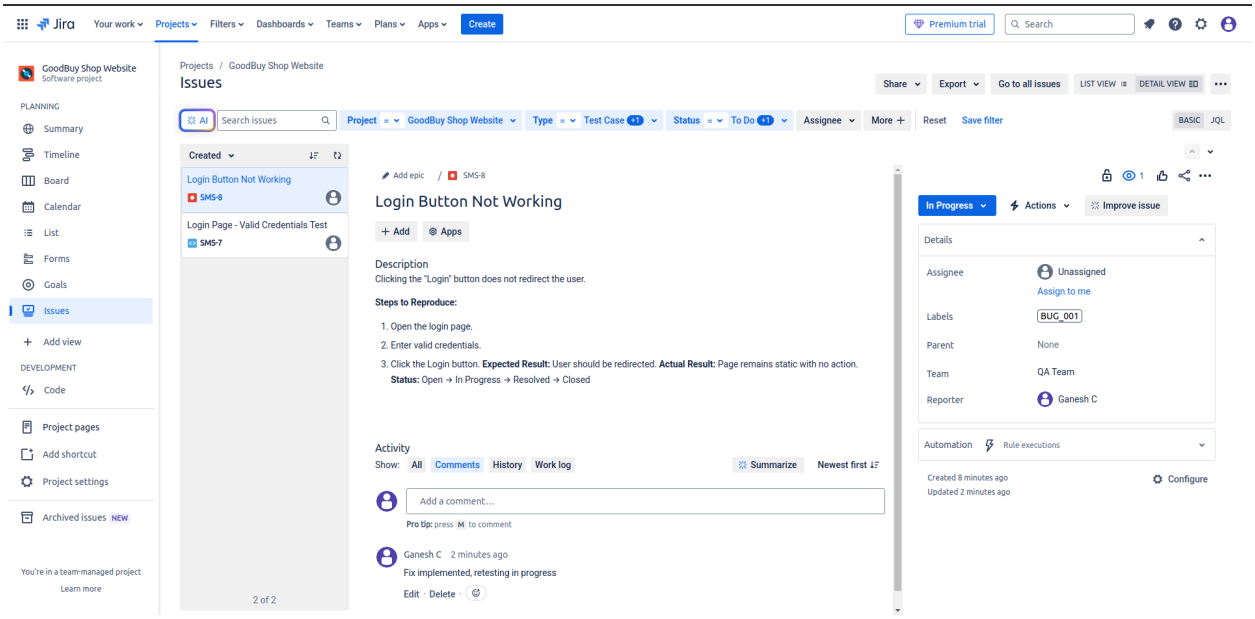
- Defect Density Report: Tracks bugs per module.
- Test Execution Report: Shows pass/fail status of test cases.
- Sprint Progress Report: Displays testing progress in an Agile sprint.

Step 7: Review & Continuous Improvement

- Conduct a retrospective to analyze testing effectiveness.
- Implement lessons learned for future QA planning.
- Update JIRA workflows for better test management.

Output:





Result:

To building a Quality Assurance Plan for a sample project using JIRA to track and manage testing activities, defects, and project quality was successfully implemented.

Ex No: 3 WRITING AND EXECUTING BASIC TEST CASES FOR A SIMPLE APPLICATION
DATE:

AIM:

To write and execute basic test cases for a simple application.

PROCEDURE:

Step 1: Identify a Sample Application For this experiment,

Consider a simple Calculator application that performs basic arithmetic operations (addition, subtraction, multiplication, and division).

Step 2: Writing Test Cases

Design test cases to verify the functionality of the Calculator application.

Test Case No.	Test Scenario	Input	Expected Output	Actual Output	Status (Pass/Fail)
TC_01	Addition of two positive numbers	5, 3	8	8	Pass
TC_02	Subtraction of two numbers	5, 3	2	2	Pass
TC_03	Multiplication of two numbers	5, 3	15	15	Pass
TC_04	Division of two numbers	6, 3	2.0	2.0	Pass
TC_05	Division by zero	6, 0	Exception	Exception	Pass

Step 3: Executing the Test Cases

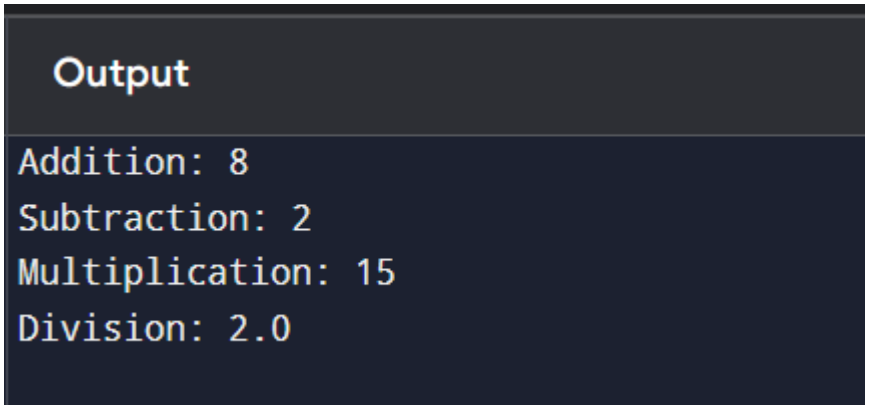
1. Run the Calculator program in a Python environment.
2. Observe the output for each test case.
3. Compare actual output with expected output.
4. Document the results in the table.

CODE IMPLEMENTATION

```
def add(a, b):
    return a + b
def subtract(a, b):
    return a - b
def multiply(a, b):
    return a * b
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

```
if __name__ == "__main__":
    print("Addition:", add(5, 3))  # Expected: 8
    print("Subtraction:", subtract(5, 3))  # Expected: 2
    print("Multiplication:", multiply(5, 3))  # Expected: 15
    print("Division:", divide(6, 3))  # Expected: 2.0
```

OUTPUT



RESULT:

Thus, basic test cases for a simple application has been successfully wrote and executed

Ex No: 4 EXECUTING TEST CASES MANUALLY ON A SAMPLE APPLICATION

DATE:

AIM:

To execute test cases manually on a sample application.

PROCEDURE:

Step 1: Selecting a Sample Application

Use the Calculator application developed in the previous experiment with minor modification to accept manual insertion of test cases.

Step 2: Identifying Test Cases

Test cases are defined as follows:

Test Case No.	Test Scenario	Input	Expected Output	Actual Output	Status (Pass/Fail)
TC_01	Addition of two positive numbers	5, 3	8.0	8.0	Pass
TC_02	Subtraction of two numbers	5, 3	2.0	2.0	Pass
TC_03	Multiplication of two numbers	5, 3	15.0	15.0	Pass
TC_04	Division of two numbers	6, 3	2.0	2.0	Pass
TC_05	Division by zero	6, 0	ERROR	ERROR	Pass
TC_06	Addition with large numbers	1e9, 1e9	2000000000.0	2000000000.0	Pass
TC_07	Subtraction resulting in zero	100, 100	0.0	0.0	Pass
TC_08	Multiplication by zero	500, 0	0.0	0.0	Pass
TC_09	Floating-point precision check	0.1, 0.2	0.30000000000000004	0.3	Pass

Step 3: Manually Executing the Test Cases

1. Open a terminal or command prompt.
2. Run the script using python calculator.py.
3. Observe the output and verify it against expected results.
4. Document the actual outputs.

CODE IMPLEMENTATION

```
def add(a, b):
    return a + b
def subtract(a, b):
```

```
    return a - b
def multiply(a, b):
    return a * b
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
def run_manual_tests():
    while True:
        operation = input("Enter operation (add, subtract, multiply, divide) or 'exit' to quit: ").strip().lower()
        if operation == 'exit':
            break
        try:
            a = float(input("Enter first number: "))
            b = float(input("Enter second number: "))
            if operation == 'add':
                print("Result:", add(a, b))
            elif operation == 'subtract':
                print("Result:", subtract(a, b))
            elif operation == 'multiply':
                print("Result:", multiply(a, b))
            elif operation == 'divide':
                print("Result:", divide(a, b))
            else:
                print("Invalid operation!")
        except ValueError as e:
            print("Error:", e)
if __name__ == "__main__":
    run_manual_tests()
```

OUTPUT

```
Output
Enter operation (add, subtract, multiply, divide) or 'exit' to quit: add
Enter first number: 5
Enter second number: 3
Result: 8.0
Enter operation (add, subtract, multiply, divide) or 'exit' to quit: subtract
Enter first number: 5
Enter second number: 3
Result: 2.0
Enter operation (add, subtract, multiply, divide) or 'exit' to quit: multiply
Enter first number: 5
Enter second number: 3
Result: 15.0
Enter operation (add, subtract, multiply, divide) or 'exit' to quit: divide
Enter first number: 6
Enter second number: 3
Result: 2.0
Enter operation (add, subtract, multiply, divide) or 'exit' to quit: divide
Enter first number: 6
Enter second number: 0
ERROR!
Error: Cannot divide by zero
Enter operation (add, subtract, multiply, divide) or 'exit' to quit: add
Enter first number: 1e9
Enter second number: 1e9
Result: 2000000000.0
Enter operation (add, subtract, multiply, divide) or 'exit' to quit: subtract
Enter first number: 100
Enter second number: 100
Result: 0.0
Enter operation (add, subtract, multiply, divide) or 'exit' to quit: multiply
Enter first number: 500
Enter second number: 0
Result: 0.0
Enter operation (add, subtract, multiply, divide) or 'exit' to quit: add
Enter first number: 0.1
Enter second number: 0.2
Result: 0.30000000000000004
```

RESULT:

Thus, Manually Executing Test Case on a sample program has been executed and the output has been verified.

5. Introduction to test automation tools – setting up and running basic automated tests.

Aim:

To understand the basics of test automation tools by setting up and executing automated tests using Selenium WebDriver and Python. This experiment will help in learning how to create and run automated test scripts for web applications.

Procedure:

Step 1: Install Required Dependencies

1. Ensure Python 3 is installed on your system:
    - Check Python version:

python --version

    - If Python is not installed, download and install it from Python's official website.
2. Open a terminal (Command Prompt / PowerShell on Windows or Terminal on Linux).
3. Install Selenium and WebDriver Manager using pip:

- Windows/Debian/Ubuntu

pip install selenium webdriver-manager

- Arch-based

    yay -S python-selenium python-webdriver-manager

4. Ensure Google Chrome is installed:
    - Windows: Download Chrome
    - Linux (Debian/Ubuntu):

sudo apt install google-chrome-stable

    - Linux (Arch-based):

yay -S google-chrome

Step 2: Set Up Selenium WebDriver

1. Open a text editor (VS Code/Vim).
2. Create a new Python script file named test_automation.py.
3. Add the following lines at the beginning of the script:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.common.by import By
import time
```

    - This ensures that WebDriver Manager automatically installs and manages ChromeDriver for you.

Step 3: Writing and Running an Automated Test

1. Write a Selenium script to open DuckDuckGo, perform a search, and verify the result.
2. Save the script after ensuring correct indentation and syntax.
3. Open a terminal and navigate to the script's location:
    - Windows (Command Prompt / PowerShell):

```
cd C:\path\to\your\script
```

    - Linux:

```
cd /path/to/your/script
```

4. Run the script using:

```
python test_automation.py
```

5. Observe the browser opening, performing actions, and closing automatically.

Step 4: Implementing an Additional Automated Test

1. Modify the script to navigate to Wikipedia and perform an automated search.
2. Create a second test function within the same script.
3. Write Selenium commands to search for a keyword and verify results.
4. Save the modified script.
5. Execute the updated script and verify the output.

Step 5: Running Automated Tests with PyTest

1. Install PyTest (if not already installed):

- Windows/Debian/Ubuntu

```
pip install pytest
```

- Arch-based

  ```
  yay -S python-pytest
  ```

2. Modify the script to be compatible with PyTest by ensuring all test functions follow the test_ naming convention.
3. Execute the script using PyTest:

```
pytest test_automation.py
```

4. Observe the structured test results in the terminal.

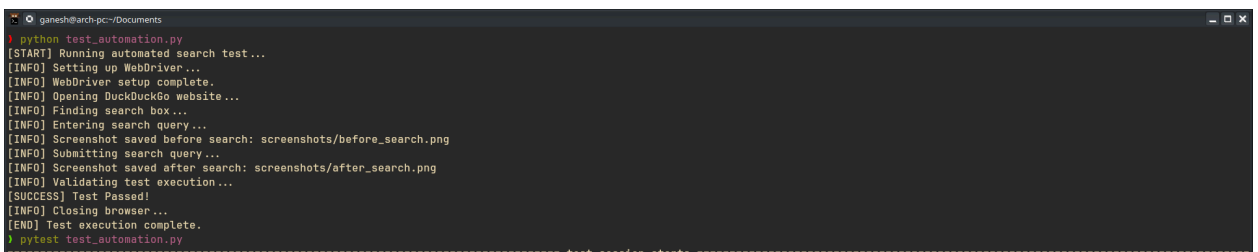
Step 6: Documenting and Analyzing Test Results

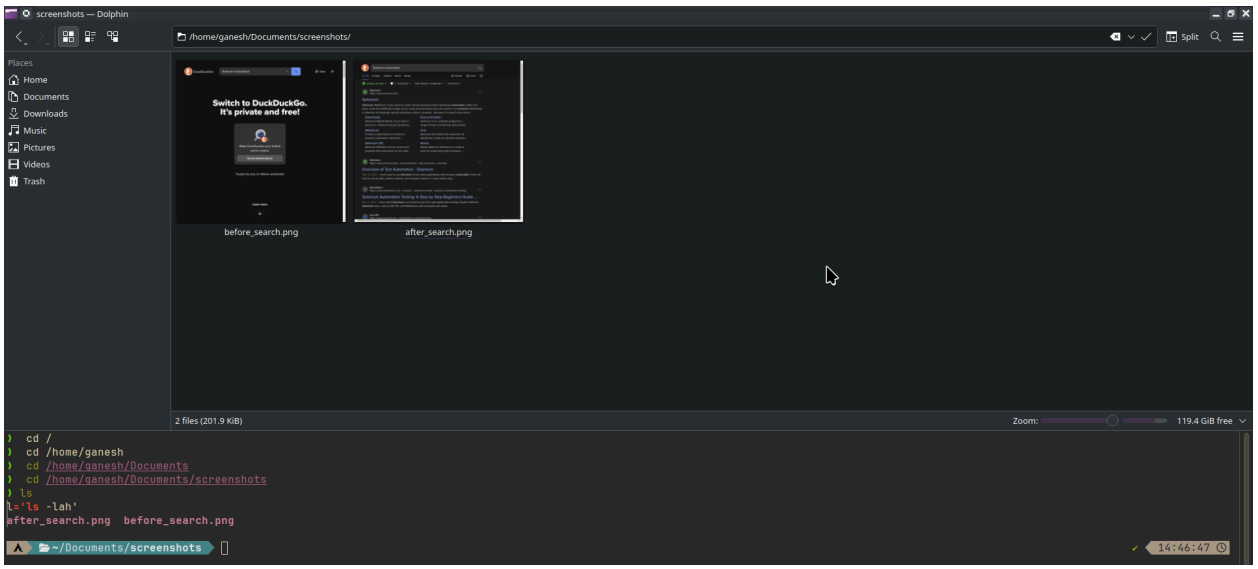
1. Record observations from both test executions.
2. Note errors or unexpected behaviors.
3. Verify if the automation achieved expected results.
4. Save logs and test reports for future reference.

Program:

```
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.common.by import By
import time
import os
# Function to ensure screenshots directory exists
def ensure_screenshot_dir():
    if not os.path.exists("screenshots"):
        os.makedirs("screenshots")
def test_duckduckgo_search():
    print("[INFO] Setting up WebDriver...")
    options = webdriver.ChromeOptions()
    service = Service(ChromeDriverManager().install())  # Auto-downloads correct ChromeDriver
    driver = webdriver.Chrome(service=service, options=options)
    print("[INFO] WebDriver setup complete.")
    print("[INFO] Opening DuckDuckGo website...")
    driver.get("https://www.duckduckgo.com")
    # Introduce delay to observe actions
    time.sleep(2)
    print("[INFO] Finding search box...")
    search_box = driver.find_element(By.NAME, "q")
    print("[INFO] Entering search query...")
    search_query = "Selenium automation"
    search_box.send_keys(search_query)
    # Capture screenshot before submitting the search
    ensure_screenshot_dir()
    screenshot_before = "screenshots/before_search.png"
    driver.save_screenshot(screenshot_before)
    print(f"[INFO] Screenshot saved before search: {screenshot_before}")
    print("[INFO] Submitting search query...")
    search_box.submit()
    # Introduce delay to observe the search results loading
    time.sleep(5)
    # Capture screenshot after search execution
    screenshot_after = "screenshots/after_search.png"
    driver.save_screenshot(screenshot_after)
    print(f"[INFO] Screenshot saved after search: {screenshot_after}")
    print("[INFO] Validating test execution...")
    assert "Selenium" in driver.title, "Test Failed"
    print("[SUCCESS] Test Passed!")
    print("[INFO] Closing browser...")
    driver.quit()
if __name__ == "__main__":
    print("[START] Running automated search test...")
    test_duckduckgo_search()
    print("[END] Test execution complete.")
```

Output:





Result:

The experiment successfully demonstrated how to set up and run automated tests using Selenium WebDriver. The script navigated to different webpages, performed searches, validated results, and closed the browser, achieving the intended objectives.

6) **Simulating the defect life cycle using a bug tracking tool**

Aim

To simulate the defect life cycle using a bug tracking tool (JIRA) and understand the different stages of a defect from creation to closure.

Procedure

1. **Log in to JIRA:**
   - Access the JIRA dashboard using valid credentials.
   - Ensure that you have the necessary permissions to create and manage issue

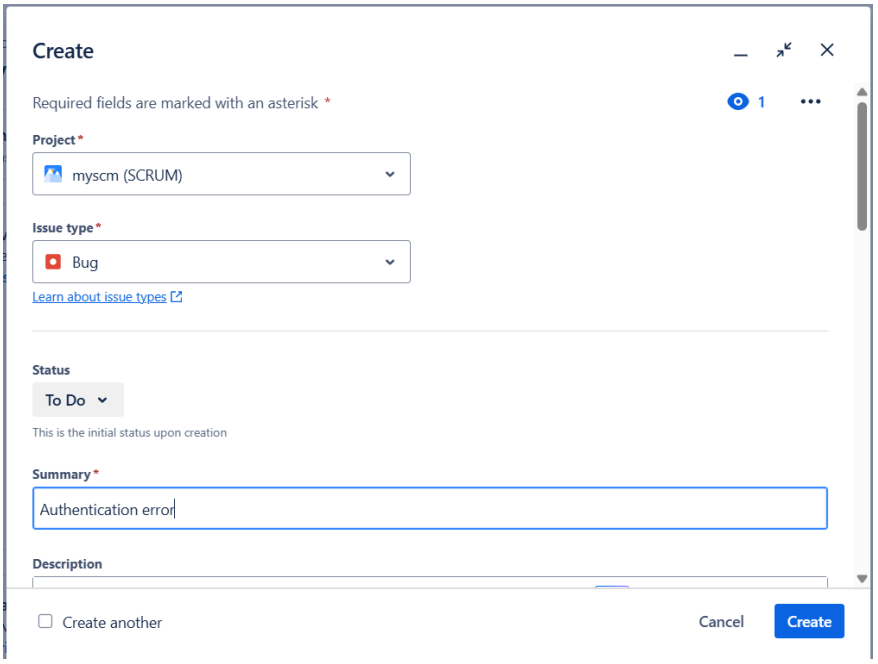
2. **Create a New Project (if needed):**
   - Navigate to "Projects" > "Create Project."
   - Choose a software development template (Scrum or Kanban).
   - Set up necessary workflows, issue types, and roles.

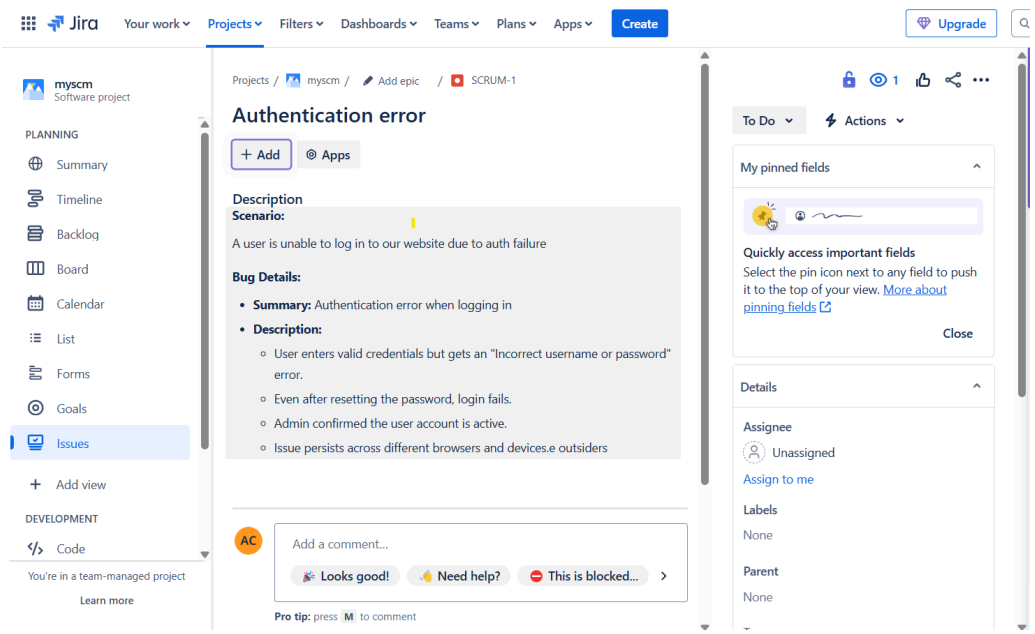
3. **Create a Bug:**
   - Click on "Create" in the JIRA dashboard.
   - Select "Bug" as the issue type.
   - Provide necessary details such as:
     - **Summary:** Short description of the defect.
     - **Description:** Detailed steps to reproduce the issue.
     - **Priority:** (Low, Medium, High, Critical).
     - **Severity:** (Minor, Major, Critical, Blocker).
     - **Environment:** Specify OS, browser, or application version.
     - **Assignee:** Assign the defect to a developer.

4. **Defect Life Cycle Simulation:**
   - **New:** Bug is created and logged in JIRA.
   - **Assigned:** The bug is assigned to the developer for analysis.
   - **In Progress:** The developer starts working on the bug fix.
   - **Fixed:** The developer resolves the defect and updates the status.
   - **Ready for Testing:** The defect is moved to testing for validation.
   - **Reopened (if necessary):** If the issue persists, it is reopened for further investigation.
   - **Closed:** If the bug is verified as fixed, it is marked as closed.

5. **Bug Verification & Closure :**
   - The tester verifies the fix by retesting the defect.
   - If resolved, the tester marks the defect as "Closed."
   - If not fixed, the bug is "Reopened" and reassigned to the developer.

Output :





Result :

The defect life cycle was successfully simulated using JIRA, demonstrating the various stages of defect tracking from identification to closure.

7) Root Cause Analysis (RCA) and Corrective Action for Identified Defects

AIM:

To analyze software defects using Root Cause Analysis (RCA) and implement corrective actions to improve software quality and reliability.

SOFWARE & HARDWARE REQUIREMENTS:

Software:

- JIRA (Bug Tracking Tool)
- Selenium (Automated Testing)
- Python
- Postman (API Testing)
- JMeter (Performance Testing)

PROCEDURE:

Step 1: Identifying Defects

1. Conduct manual and automated testing using Selenium and Postman.
2. Log defects in JIRA with detailed descriptions, screenshots, and replication steps.
3. Categorize defects based on severity (Critical, Major, Minor) and priority (High, Medium, Low).

Step 2: Root Cause Analysis (RCA)

1. Use the **5 Whys Analysis** technique:
    - Why did the defect occur?
    - Why was it not detected earlier?
    - Why did testing miss the issue?
    - Why was the requirement unclear?
    - Why did the development process fail?
2. Use the **Fishbone Diagram (Ishikawa)** to classify the root cause under categories:
    - **People:** Lack of training, miscommunication.
    - **Process:** Incorrect workflow, missing validation steps.
    - **Technology:** Outdated libraries, API failures.
    - **Environment:** Server misconfiguration, database issues.

Step 3: Implementing Corrective Actions

1. **Code Fixes:**
    - Modify affected code and test locally before committing.
    - Use GitHub branching strategy (feature, dev, main) to track fixes.
2. **Process Improvements:**
    - Strengthen code review processes using SonarQube.
    - Implement unit tests for early defect detection.
3. **Test Coverage Expansion:**
    - Increase UI and API test coverage using Selenium and Postman.
    - Automate regression testing.
4. **Monitoring and Logging Enhancements:**

- Implement better logging mechanisms using Log4j or ELK stack.
- Use Prometheus and Grafana for real-time monitoring.

Step 4: Validating Corrective Actions

1. Retest fixed defects in JIRA and update the status.
2. Conduct impact analysis to ensure new changes do not introduce additional issues.
3. Perform load and stress testing using JMeter.

OUTPUT:

1. Identified and categorized software defects.
2. Conducted RCA using **5 Whys Analysis** and **Fishbone Diagram**.
3. Implemented corrective actions and validated fixes.
4. Improved testing strategies and defect prevention techniques.

RESULT:

Thus, Root Cause Analysis (RCA) and Corrective Actions for identified defects were successfully performed, leading to improved software quality and defect prevention strategies.

Ex No: 8 CONDUCTING PERFORMANCE TESTS USING A PERFORMANCE TESTING TOOL DATE:

AIM:

To test the performance of the application using performance testing tool JMeter.

PROCEDURE:

Procedure for Using JMeter for Web Application Testing

1. Setting Up JMeter

Download and Install:

- Download the JMeter archive from the [Apache JMeter website](https://jmeter.apache.org/).
- Extract the archive to a desired location. Start JMeter:
- Navigate to the `bin` directory within the extracted archive.
- Run `jmeter.bat` (Windows) or `jmeter.sh` (Linux/macOS) to launch the GUI. Ensure Java is Installed:
- JMeter requires Java to run, so ensure a compatible version of Java is installed.

2. Creating a Test Plan

New Test Plan:

- Open JMeter and create a new test plan by navigating to File -> New -> Test Plan . Rename Test Plan:
- Rename the test plan for better organization.

3. Adding a Thread Group

Right-click on the Test Plan:

- Right-click on the test plan in the tree view and select Add -> Threads (Users) -> Thread Group .

Configure Thread Group:

- Number of Threads (Users): Set the number of virtual users to simulate.
- Ramp-Up Period (seconds): Define the time it takes for all threads to start.
- Loop Count: Specify how many times each thread will execute the test.
- Other Options: Configure additional options like delays, test start and stop times, and actions to take after a sampler error.

4. Adding HTTP Requests (Samplers) Right-click on the Thread Group:

- Select Add -> Sampler -> HTTP Request .

Configure HTTP Request:

- Protocol: Select the protocol (e.g., HTTP, HTTPS).
- Server Name or IP: Enter the server name or IP address.
- Port: Specify the port number (e.g., 80 for HTTP, 443 for HTTPS).
- Path: Enter the path to the resource.
- Method: Select the HTTP method (e.g., GET, POST).
- Parameters: Add any necessary parameters for the request.

5. Adding Listeners

Right-click on the Thread Group:

- Select Add -> Listener .

Choose a Listener:

- Select a listener to view the results, such as:
  - View Results in Table
  - View Results in Tree
  - Aggregate

Report Configure

Listener:

- Modify the listener settings as needed.

6. Running the

Test Run the

Test:

- Click the green "Run" button or go to Run -> Run .

Analyze Results:

- Review test results in the selected listener.

7. Key JMeter Concepts

- Test Plan: The main container for your test setup.
- Thread Group: Simulates multiple users or concurrent requests.
- Sampler: Represents a single request or action in your test.
- Listener: Used to view and analyze the results of your test.
- HTTP Request Defaults: Allows you to set default values for HTTP requests, such as protocol, server, and port.

8. Advanced Features

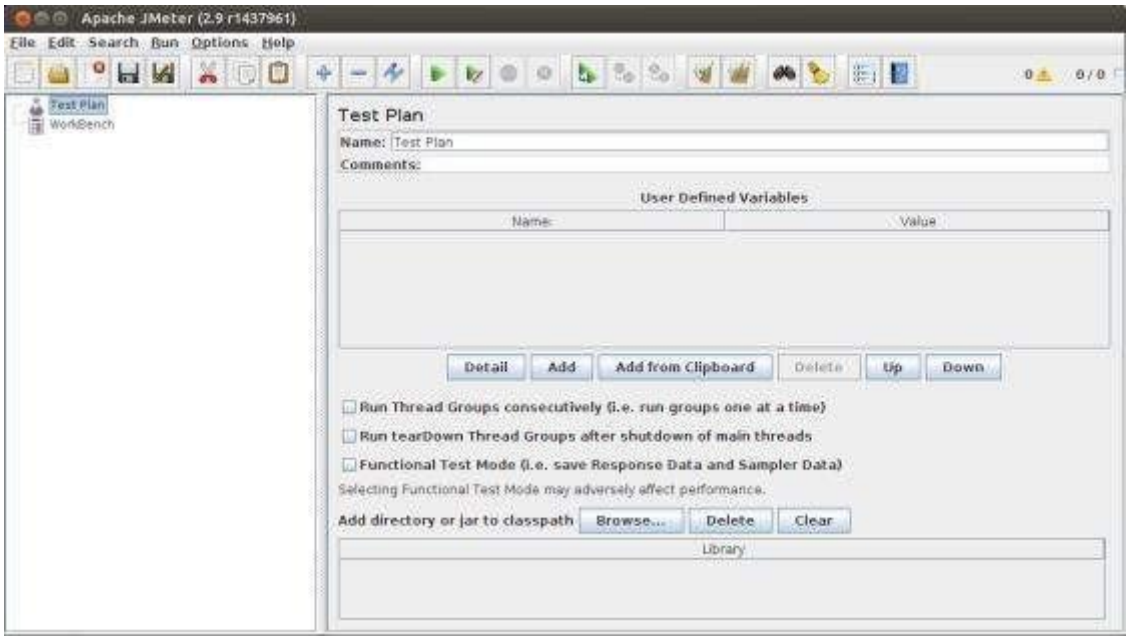
- Assertions: Used to validate the response from the server.
- Load Testing: JMeter can simulate high load and stress conditions.
- Non-GUI Mode: JMeter can also be run in command-line mode for automation and scripting.

PROGRAM:

Build a simple test plan which tests a web page.Write a test plan in Apache JMeter so that we can test the performance of the web page shown by the URL www.example.com

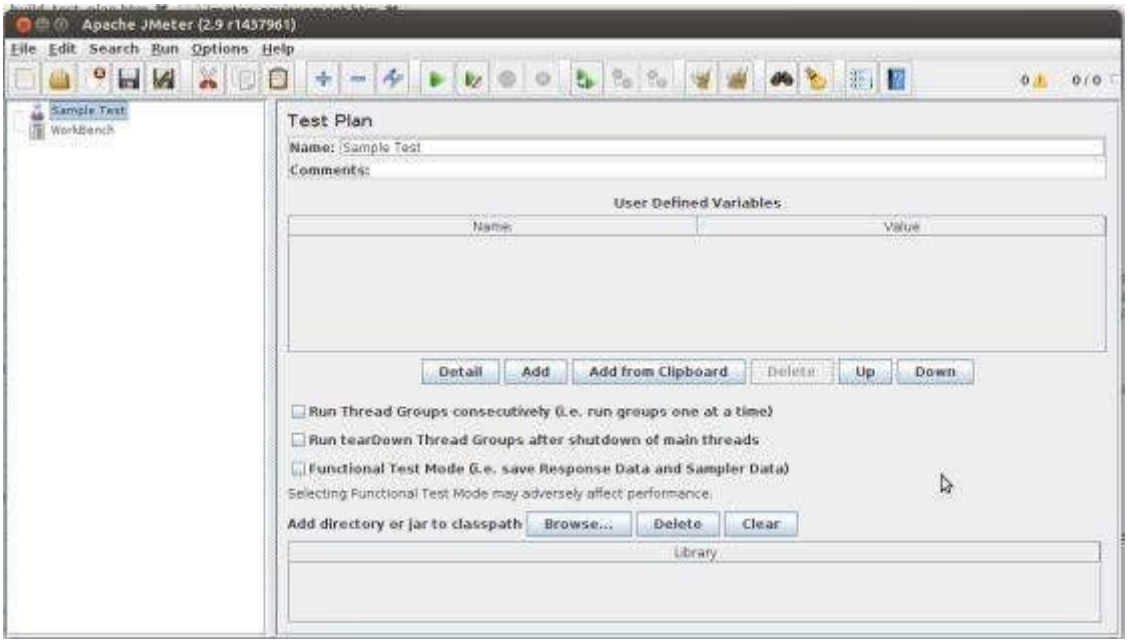
Start JMeter

Open the JMeter window by clicking on **/home/raj/apache-jmeter-2.9/bin/jmeter.sh**. The JMeter window appear as below −



Rename the Test Plan

Change the name of test plan node to *Sample Test* in the *Name* text box. To change the focus to workbench node and back to the Test Plan node to see the name getting reflected.

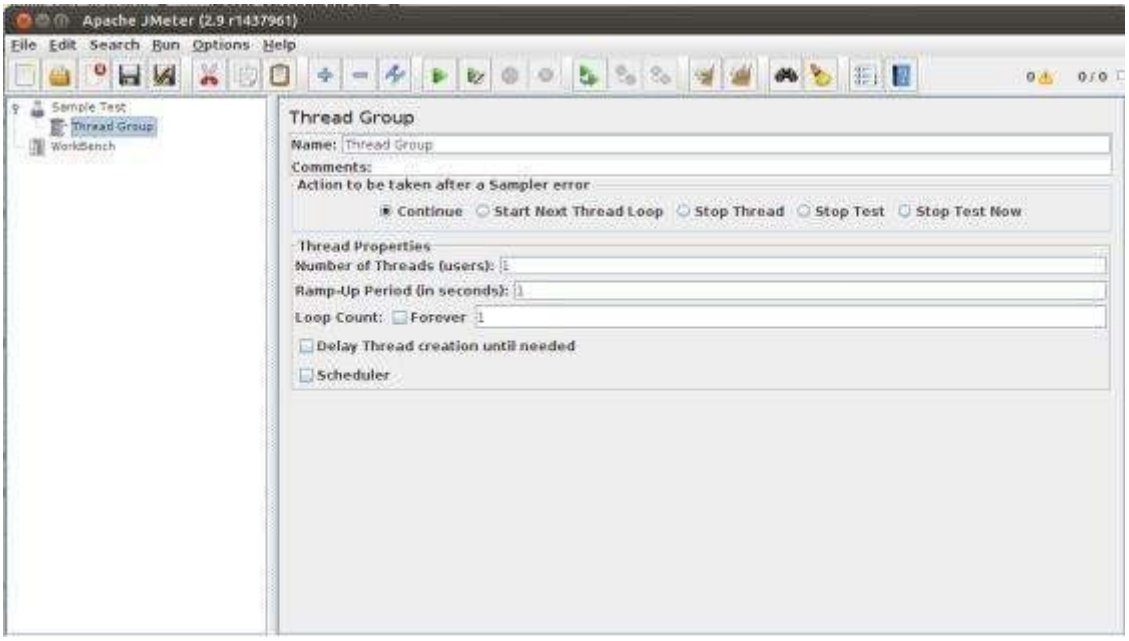


Add Thread Group

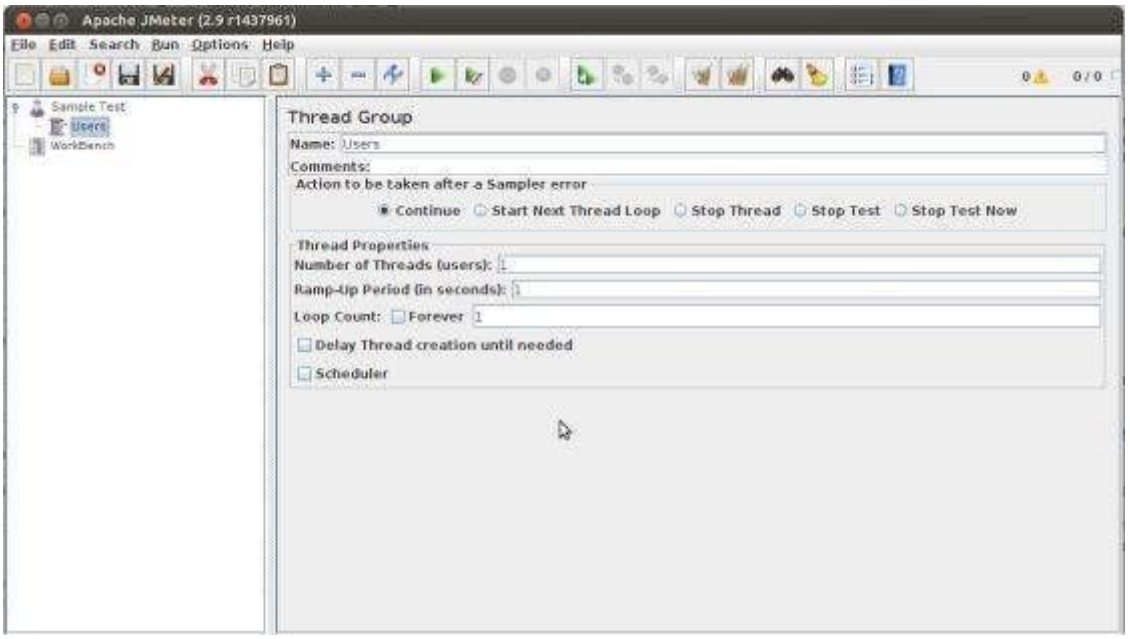
Add first element in the window. Add one Thread Group, which is a placeholder for all other elements like Samplers, Controllers, and Listeners.Configure number of users to simulate.

In JMeter, all the node elements are added by using the context menu.

- Right-click the element where you want to add a child element node.
- Choose the appropriate option to add.
- Right-click on the Sample Test *ourTestPlan* > Add > Threads *Users* > Thread Group. Thus, the Thread Group gets added under the Test Plan *SampleTest* node.



Name the Thread Group as *Users*. For us, this element means users visiting the TutorialsPoint Home Page.

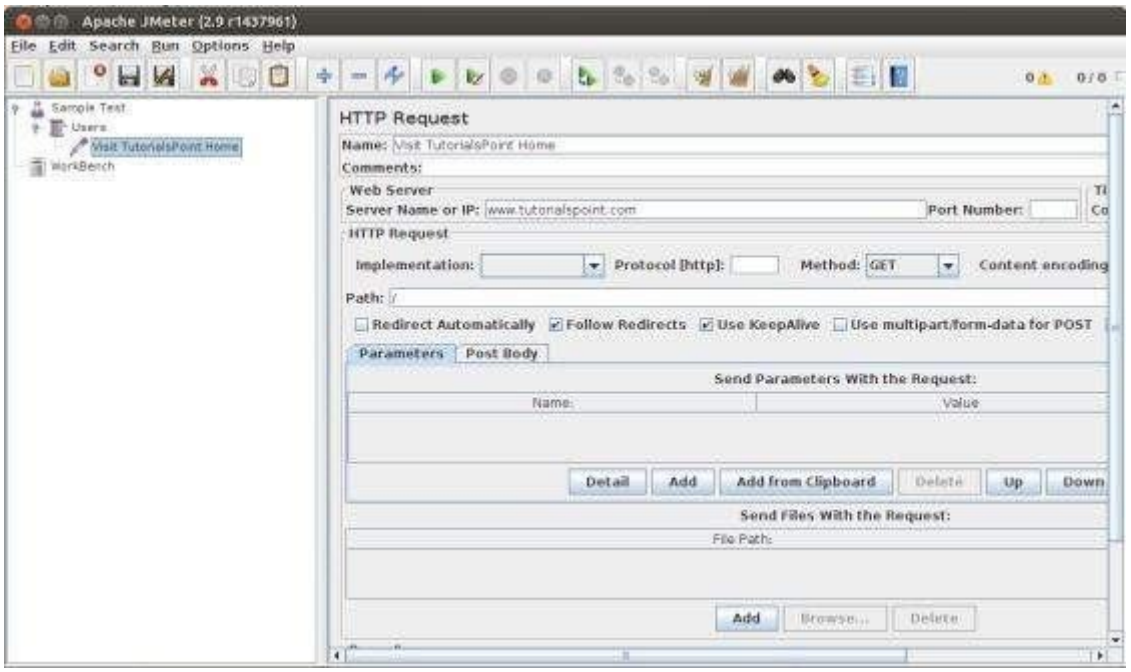


Add Sampler

Add one Sampler in our Thread Group *Users*. For adding Thread group, this time we will open the context

menu of the Thread Group *Users* node by right-clicking and It will add one empty HTTP Request Sampler under the Thread Group *Users* node. Let us configure this node element −

add HTTP Request Sampler by choosing Add > Sampler > HTTP request option.

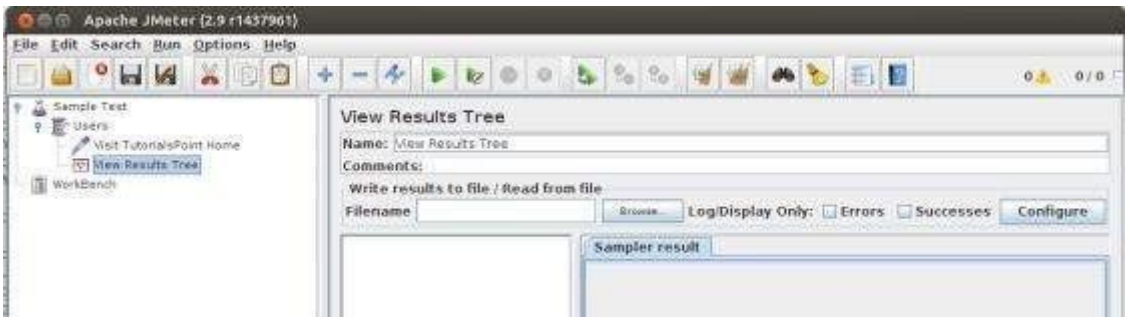


Add Listener

We will now add a listener. Let us add View Results Tree Listener under the Thread Group *User* node. It will ensure that the results of the Sampler will be available to view in this Listener node element.

To add a listener −

- Open the context menu
- Right-click the Thread Group *Users*
- Choose Add > Listener > View Results Tree option

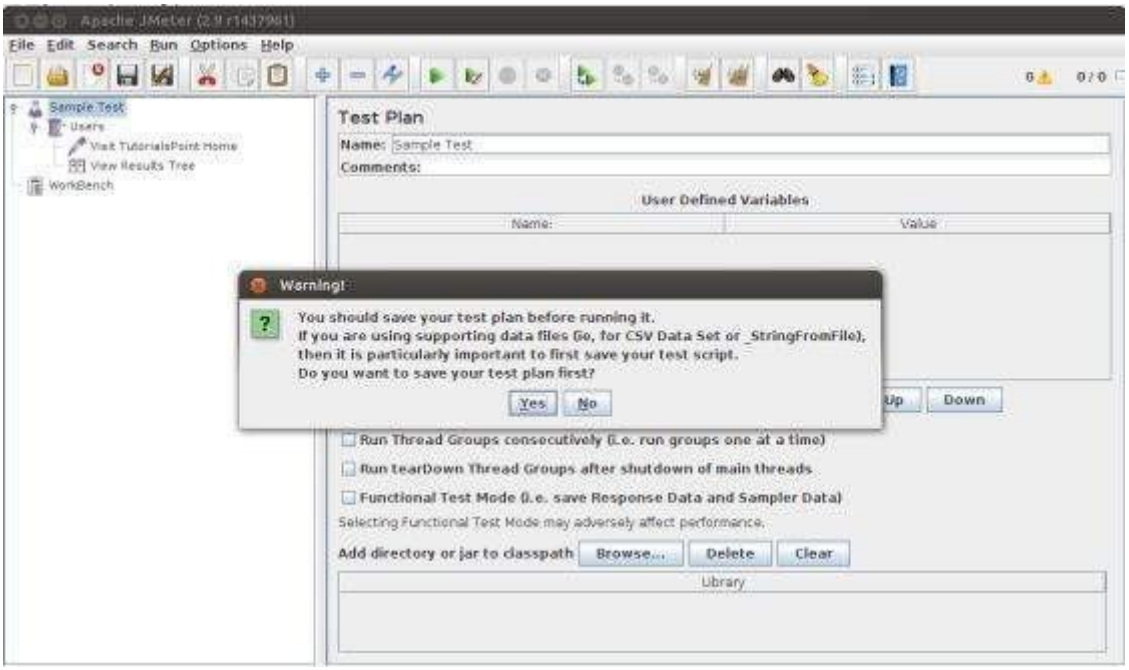


Run the Test Plan

Now with all the setup, let us execute the test plan. With the configuration of the Thread Group *Users*, we keep all the default values. It means JMeter will execute the sampler only once. It is similar to a single user, only once.

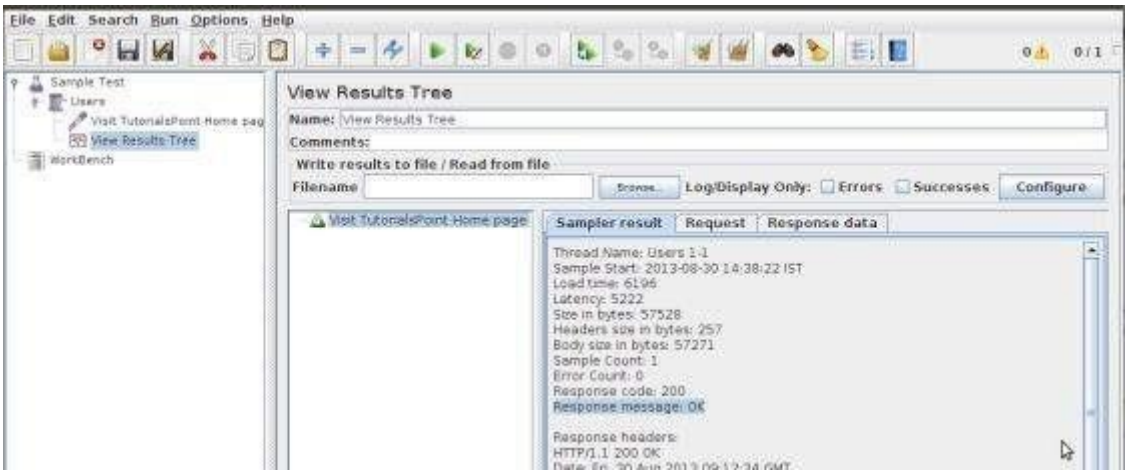
This is similar to a user visiting a web page through browser, with JMeter sampler. To execute the test plan, Select Run from the menu and select Start option.

Apache JMeter asks us to save the test plan in a disk file before actually starting the test. This is important if you want to run the test plan multiple times. You can opt for running it without saving too.



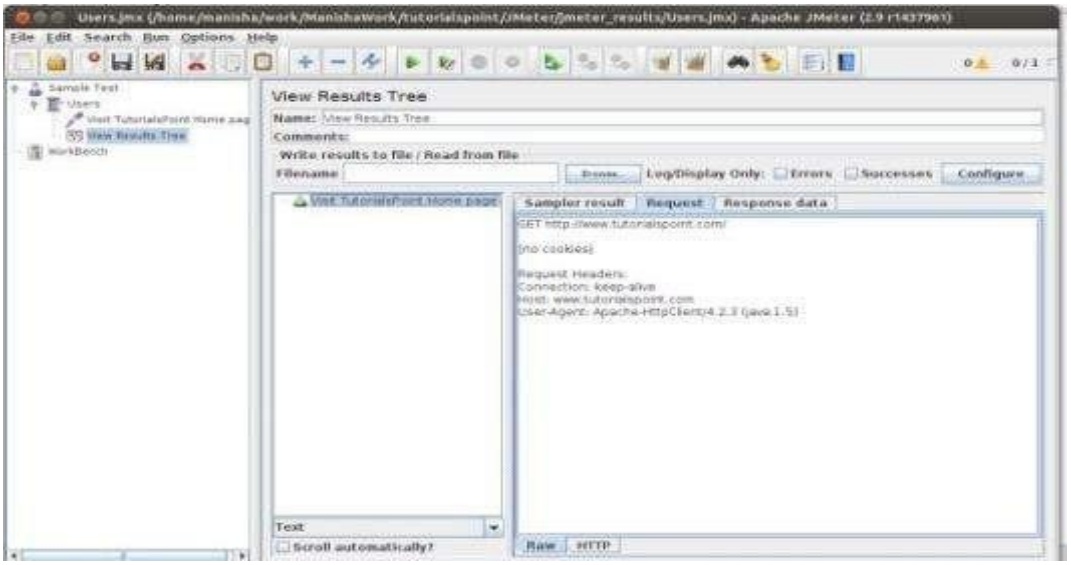
View the Output

Keep the setting of the thread group as single thread *one user only* and loop for 1 time *run only one time*, hence we will get the result of one single transaction in the View Result Tree Listener.

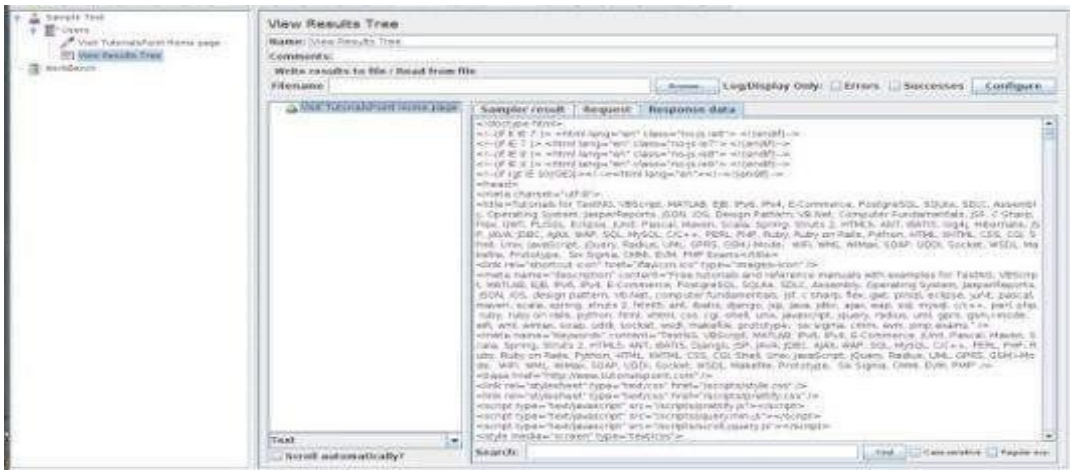


Details of the above result are −

- Green color against the name *Visit example Home Page* indicates success.
- JMeter has stored all the headers and the responses sent by the web server and ready to show us the result in many ways.
- The first tab is Sampler Results. It shows JMeter data as well as data returned by the web server.
- The second tab is Request, which shows all the data sent to the web server as part of the request.



The last tab is Response data. In this tab, the listener shows the data received from server in text format.



This is just a simple test plan which executes only one request. But JMeter's real strength is in sending the same request, as if many users are sending it. To test the web servers with multiple users, we need to change the Thread Group *Users* settings.

RESULT:

Thus we can test the performance of the application using the performance test tool

Ex No: 9 IDENTIFYING AND ADDRESSING SECURITY VULNERABILITIES IN A SAMPLE APPLICATION

DATE:

AIM:

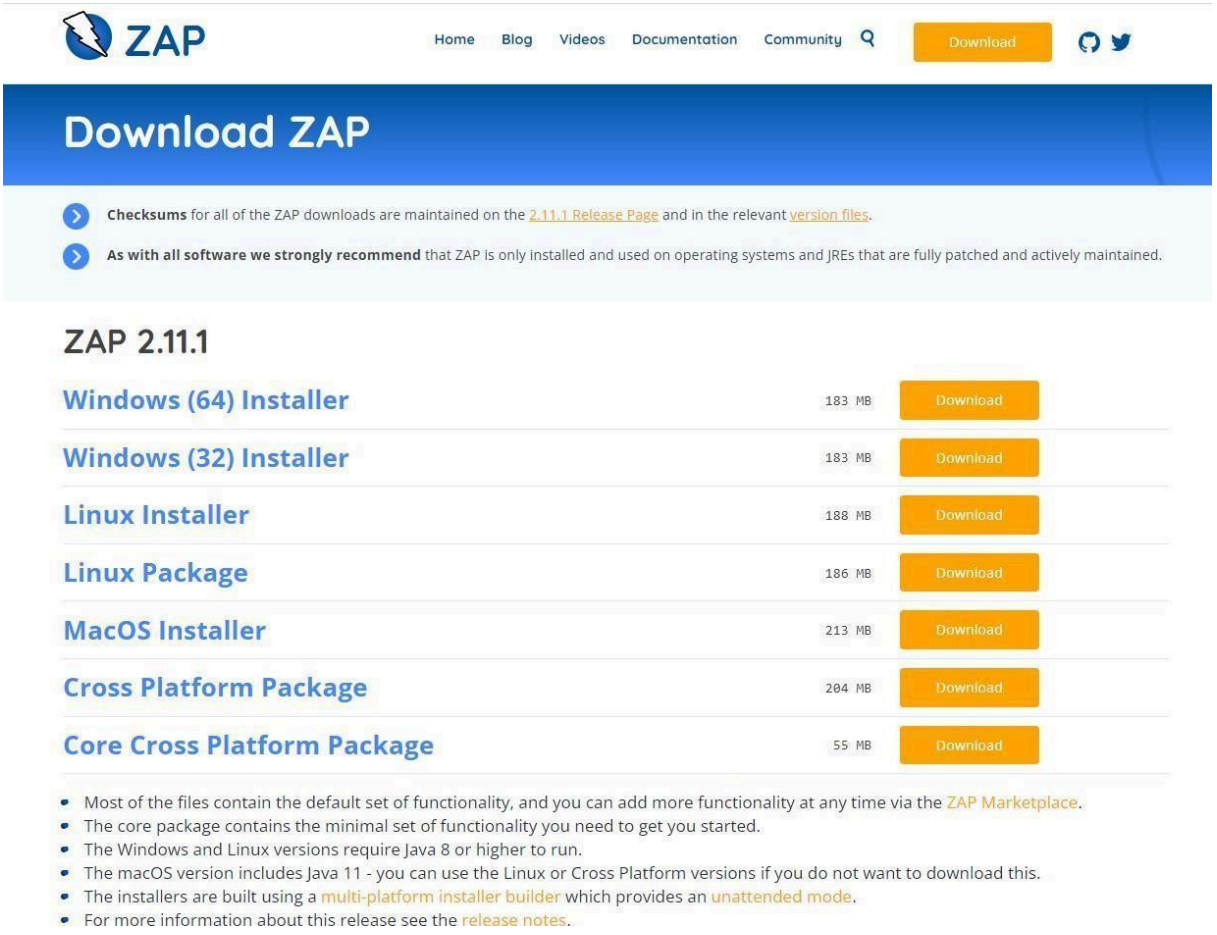
To identify and address the security vulnerabilities in a sample application.

PROCEDURE:

1. **Installing ZAP**

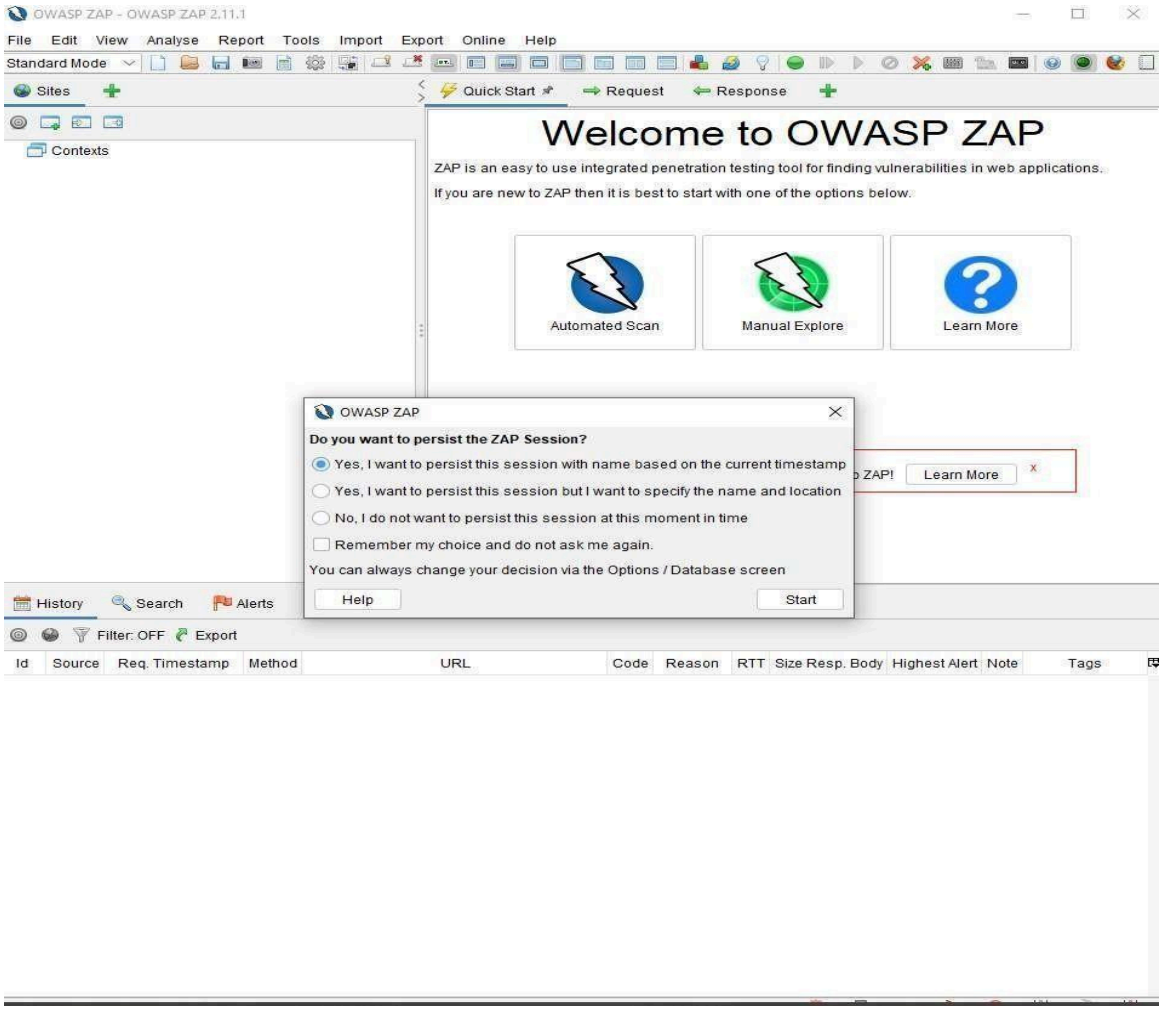
You can download the latest version from the OWASP ZAP website for your operating system to install ZAP or reference the ZAP docs for a more detailed installation guide.

-

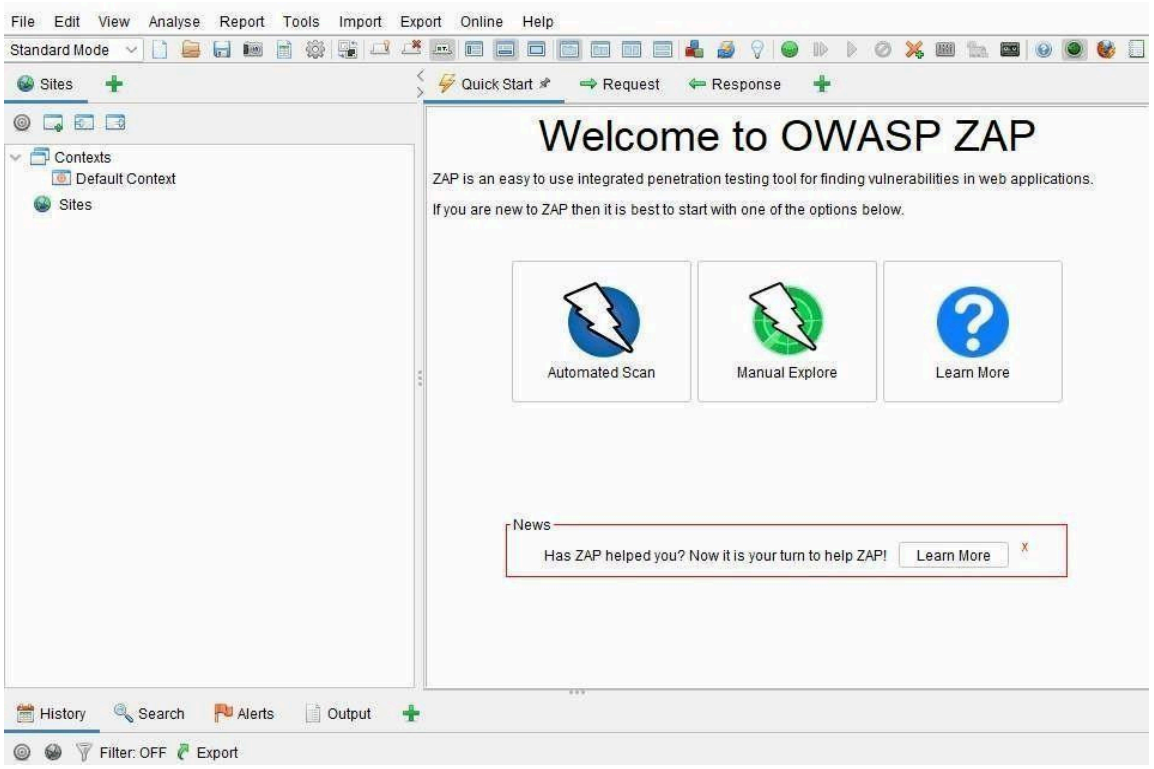


2. **Persisting a session**

Persisting a session in OWASP ZAP means that the session will be saved and can be reopened at a later time. This is useful if you want to continue testing a website or application at a later time.



The prompt gives two options to persist in the session. You can use the default to name the session based on the current timestamp or set your name and location.
Alternatively, you can persist a session by going to ‘File’ and choosing ‘Persist Session…’. give the session a name and click on the ‘Save’ button.



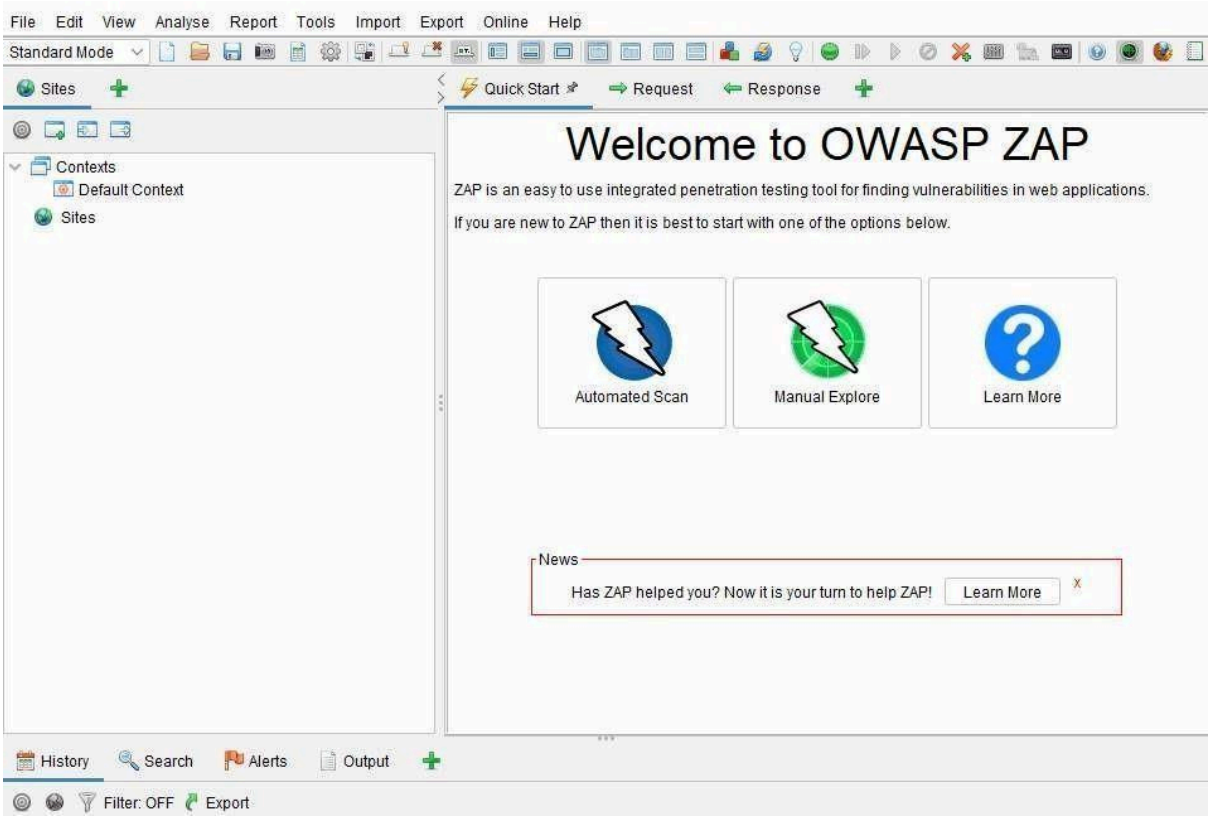
3. **Running an automated scan**

Running an automated scan in OWASP ZAP is a way to check for common security vulnerabilities in web applications. This is done by sending requests to the application and analyzing the responses for signs of common vulnerabilities. It can help to find security issues early in the development process before they are exploited.

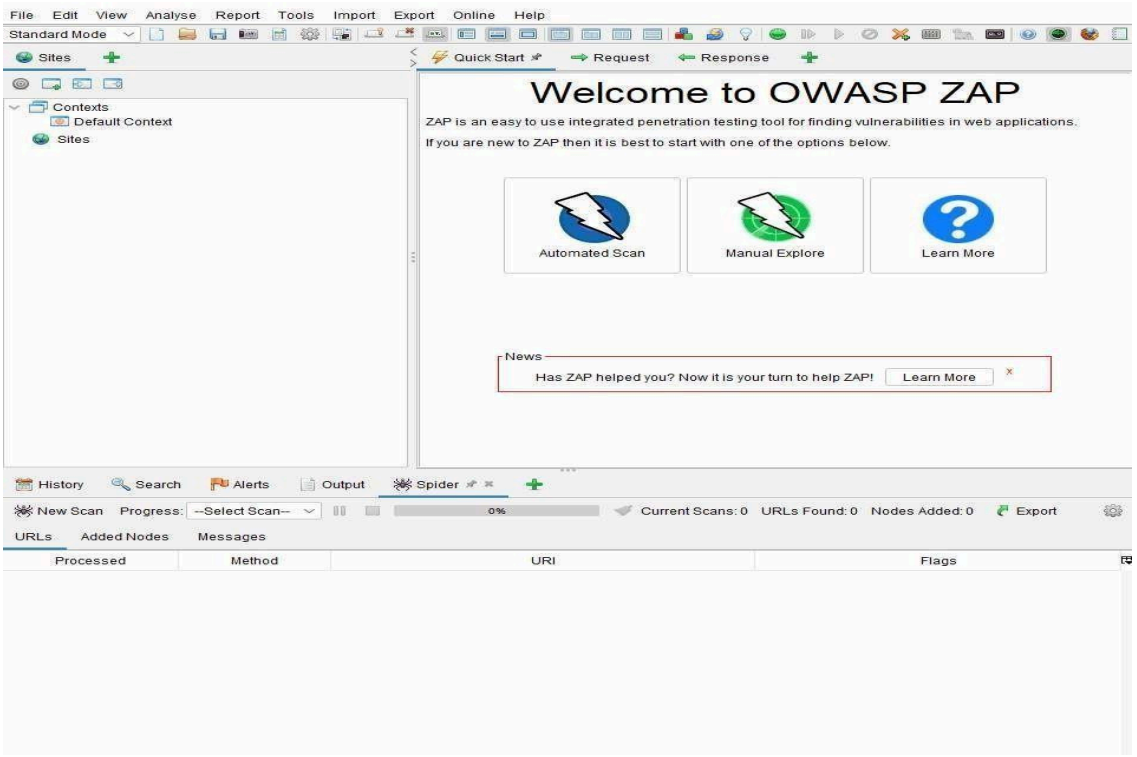
With OWASP ZAP, you can use a ZAP spider or the AJAX spider. So what's the difference?

ZAP spider is a web crawler that can automatically find security vulnerabilities in web applications. Meanwhile, the AJAX spider is a web crawler designed to crawl and attack AJAX-based web applications.

Clicking on the 'Tools' option will give you a list of available pentesting tools provided by OWASP ZAP.

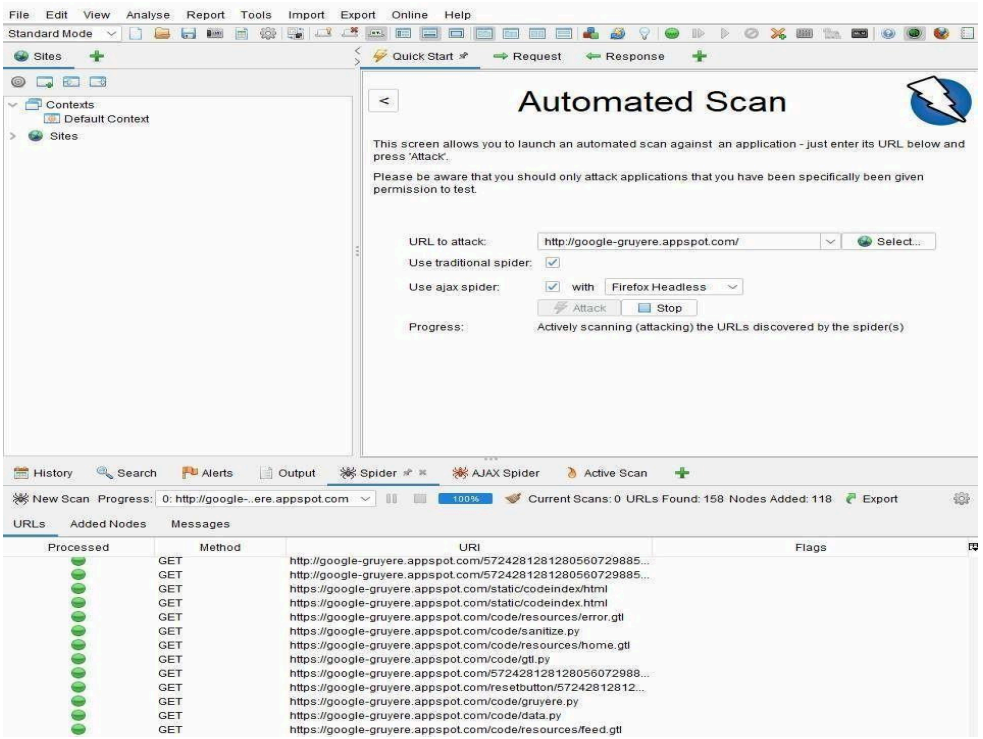


To run an automated scan, you can use the quick start "Automated Scan" option under the "Quick Start" tab. Enter the URL of the site you want to scan in the "URL to attack" field, and then click "Attack!"


File Edit View Analyse Report Tools Import Export Online Help
Standard Mode
Sites
Quick Start
Request
Response
Contexts
Default Context
Sites
Welcome to OWASP ZAP
ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications.
If you are new to ZAP then it is best to start with one of the options below.
Automated Scan
Manual Explore
Learn More
News
Has ZAP helped you? Now it is your turn to help ZAP!
Learn More
History
Search
Alerts
Output
Spider
New Scan
Progress:
--Select Scan--
0%
Current Scans: 0
URLs Found: 0
Nodes Added: 0
Export
URLs
Added Nodes
Messages
Processed
Method
URI
Flags

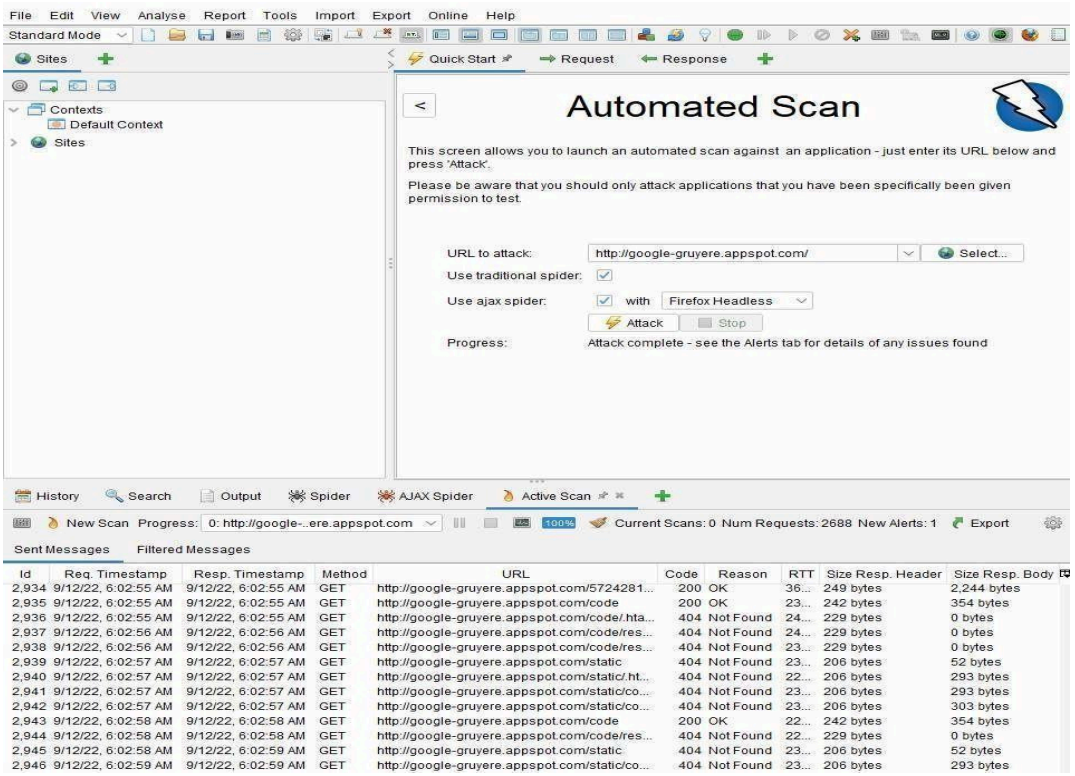
4. **Interpreting test results**

Interpreting test results in OWASP ZAP is vital to understand the scan findings and determine which issues require further investigation. Additionally, it can help to prioritize remediation efforts. In OWASP ZAP, you can view alerts by clicking on the "Alerts" tab. This tab will show you a list of all the alerts that have been triggered during your testing. The alerts are sorted by risk level, with the highest risk alerts at the top of the list. OWASP ZAP will give details of the discovered vulnerabilities and suggestions on how you can fix them.



5. **Viewing alerts and alert details**

Viewing alerts and alert details in OWASP ZAP is a way to see what potential security issues have been identified on a website. It can help security and administrators understand what needs to be fixed to improve the app's security. If you cannot find your ‘Alerts’ tab, you can access it via the ‘View’ menu, along with other options available in OWASP ZAP. Once you have your ‘Alerts’ tab, you can navigate the various vulnerabilities discovered and explore the reports generated by OWASP ZAP.



Id	Req. Timestamp	Resp. Timestamp	Method	URL	Code	Reason	RTT	Size Resp. Header	Size Resp. Body
2,934	9/12/22, 6:02:55 AM	9/12/22, 6:02:55 AM	GET	http://google-gruyere.appspot.com/5724281...	200	OK	36...	249 bytes	2,244 bytes
2,935	9/12/22, 6:02:55 AM	9/12/22, 6:02:55 AM	GET	http://google-gruyere.appspot.com/code	200	OK	23...	242 bytes	354 bytes
2,936	9/12/22, 6:02:55 AM	9/12/22, 6:02:55 AM	GET	http://google-gruyere.appspot.com/code/.hta...	404	Not Found	24...	229 bytes	0 bytes
2,937	9/12/22, 6:02:56 AM	9/12/22, 6:02:56 AM	GET	http://google-gruyere.appspot.com/code/res...	404	Not Found	24...	229 bytes	0 bytes
2,938	9/12/22, 6:02:56 AM	9/12/22, 6:02:56 AM	GET	http://google-gruyere.appspot.com/code/res...	404	Not Found	23...	229 bytes	0 bytes
2,939	9/12/22, 6:02:57 AM	9/12/22, 6:02:57 AM	GET	http://google-gruyere.appspot.com/static	404	Not Found	23...	206 bytes	52 bytes
2,940	9/12/22, 6:02:57 AM	9/12/22, 6:02:57 AM	GET	http://google-gruyere.appspot.com/static/.ht...	404	Not Found	22...	206 bytes	293 bytes
2,941	9/12/22, 6:02:57 AM	9/12/22, 6:02:57 AM	GET	http://google-gruyere.appspot.com/static/co...	404	Not Found	23...	206 bytes	293 bytes
2,942	9/12/22, 6:02:57 AM	9/12/22, 6:02:57 AM	GET	http://google-gruyere.appspot.com/static/co...	404	Not Found	23...	206 bytes	303 bytes
2,943	9/12/22, 6:02:58 AM	9/12/22, 6:02:58 AM	GET	http://google-gruyere.appspot.com/code	200	OK	22...	242 bytes	354 bytes
2,944	9/12/22, 6:02:58 AM	9/12/22, 6:02:58 AM	GET	http://google-gruyere.appspot.com/code/res...	404	Not Found	22...	229 bytes	0 bytes
2,945	9/12/22, 6:02:58 AM	9/12/22, 6:02:59 AM	GET	http://google-gruyere.appspot.com/static	404	Not Found	23...	206 bytes	52 bytes
2,946	9/12/22, 6:02:59 AM	9/12/22, 6:02:59 AM	GET	http://google-gruyere.appspot.com/static/co...	404	Not Found	23...	206 bytes	293 bytes

6. **Exploring an application manually**

Exploring an application manually in OWASP ZAP is a process of manually testing the application for security vulnerabilities. It is done to identify any potential security risks that may be present in the application. Doing this can help ensure that the application is as secure as possible.

The manual scan complements the automated scan by providing a more in-depth analysis of the application and allowing you to navigate the pentest process. The automated scan may miss some vulnerabilities, but the manual scan may pick up missed issues. However, the manual scan can be time-consuming and may not be feasible for large applications.

To explore an application manually, select “Manual Explore.” Select your browser, and OWASP ZAP will launch a proxy in your browser. Here, you will be given pentesting tools such as spiders, and if a vulnerability is discovered, an alert flag will be added to the alerts panel.

RESULT:

Thus we can identify and address the security vulnerabilities in a sample application.