Problem Statement 1:

Automated Testing:

1.● Write a simple test script (using a tool of your choice) to verify the integration between the frontend and backend services.

● The test should check that the frontend correctly displays the message returned by the backend.?

Answer

To create a test script using Selenium with Python and the unittest framework to verify the integration between frontend and backend services, you'll need to follow these steps:

1. **Set Up Your Environment**: Ensure you have Selenium and unittest installed. You can install Selenium via pip if you haven't already:

   Command -- pip install selenium

   2. **WebDriver Setup**: Make sure you have the appropriate WebDriver installed (e.g., ChromeDriver for Google Chrome, GeckoDriver for Firefox).

- **Write the Test Script**: Below is a simple example script using Selenium with the unittest framework. This script assumes you have a web application where the frontend displays a message returned by the backend service.

Code [Selenium with python using unit test framework]

```
import unittest

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.chrome.service import Service as ChromeService

from webdriver_manager.chrome import ChromeDriverManager


class TestFrontendBackendIntegration(unittest.TestCase):


    @classmethod
    def setUpClass(cls):
        # Set up the WebDriver instance
        cls.driver = webdriver.Chrome(service=ChromeService(ChromeDriverManager().install()))
        cls.driver.get('http://your-frontend-url.com')  # Replace with your frontend URL

    @classmethod
    def tearDownClass(cls):
        # Close the WebDriver instance
        cls.driver.quit()

    def test_message_displayed_correctly(self):
        # Find the element where the message should be displayed
        message_element = self.driver.find_element(By.ID, 'message-id')  # Replace with the actual
ID of the message element

        # Get the text from the element
        displayed_message = message_element.text

        # Define the expected message
        expected_message = 'Expected message from backend'  # Replace with the actual expected
message

        # Assert that the displayed message is what we expect
        self.assertEqual(displayed_message, expected_message, f'Expected message
"{expected_message}", but got "{displayed_message}"')

if __name__ == '__main__':
    unittest.main()
```

Explanation:

- **setUpClass**: Sets up the WebDriver before running any test methods.
- **tearDownClass**: Cleans up the WebDriver after all tests have run.
- **test_message_displayed_correctly**: A test method that:
    - Locates the element where the backend message should appear using find_element (adjust the selector based on your frontend structure).
    - Retrieves the text of that element.
    - Compares it with the expected message.
    - Uses assertEqual to verify that the displayed message matches the expected message.

Run the Test: Save this script as test_integration.py and run it with:

Ensure that your backend is up and running, and the frontend is correctly pointing to the backend service for the test to be valid.

Adjust the script as necessary to fit your specific frontend and backend setup, including the URL, element selectors, and expected messages.

2.To verify that the frontend service can successfully communicate with the backend service and to ensure that accessing the frontend URL displays a greeting message fetched from the backend, you can follow these steps: ?

Answer

1. **Setup Your Testing Environment**:

Make sure you have the necessary tools and dependencies installed. You will need `Selenium` for browser automation and `unittest` for running the test. Additionally, `webdriver_manager` can simplify WebDriver management

pip install selenium webdriver-manager

**```**

2. **Write the Test Script**:

The script will use Selenium to navigate to the frontend URL and verify that the greeting message is correctly displayed. Here's a complete example using Python and the `unittest` framework

python

```
import unittest
from selenium import webdriver
from selenium.webdriver.chrome.service import Service as ChromeService
from selenium.webdriver.common.by import By
from webdriver_manager.chrome import ChromeDriverManager

class TestFrontendBackendIntegration(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        # Initialize the WebDriver (using ChromeDriver in this example)
        cls.driver =
webdriver.Chrome(service=ChromeService(ChromeDriverManager().install()))
        # Navigate to the frontend URL
        cls.driver.get('http://your-frontend-url.com')  # Replace with your frontend URL

    @classmethod
    def tearDownClass(cls):
        # Close the WebDriver instance
        cls.driver.quit()

    def test_greeting_message(self):
        # Find the element where the greeting message is expected to be displayed
        message_element = self.driver.find_element(By.ID, 'greeting-message-id')  # Replace
with the actual ID or selector of the message element

        # Retrieve the text of the greeting message
        displayed_message = message_element.text
```

```
        # Define the expected greeting message
        expected_message = 'Hello, welcome to our site!'  # Replace with the actual expected greeting message

        # Assert that the displayed message matches the expected message
        self.assertEqual(displayed_message, expected_message, f'Expected greeting message "{expected_message}", but got "{displayed_message}"')

  if __name__ == '__main__':
      unittest.main()
  ```

3. **Explanation**:

  - **`setUpClass`**: Initializes the WebDriver and navigates to the frontend URL. This setup is run once before any tests are executed.

  - **`tearDownClass`**: Closes the WebDriver after all tests are complete. This ensures proper cleanup.

  - **`test_greeting_message`**: This test method:

    - Locates the element where the greeting message should be displayed. Make sure to replace `'greeting-message-id'` with the actual ID or CSS selector of the element on your frontend.

    - Retrieves the text from the located element.

    - Compares the retrieved text with the expected greeting message.

    - Asserts that the retrieved message matches the expected message, providing an error message if they do not match.

4. **Run the Test**:

  Save the script as `test_integration.py` and execute it using the command:

  python test_integration.py

Ensure that your frontend service is running and correctly configured to fetch the greeting message from the backend.

5. **Adjust the Script**:

Modify the following parts according to your application:

- **Frontend URL**: Replace `'http://your-frontend-url.com'` with the actual URL of your frontend service.

- **Element Selector**: Update `By.ID, 'greeting-message-id'` with the appropriate selector for the element displaying the greeting message (e.g., `By.CLASS_NAME`, `By.CSS_SELECTOR`).

- **Expected Message**: Replace `'Hello, welcome to our site!'` with the actual message expected from your backend service.

This script will help ensure that your frontend correctly displays the greeting message from the backend, verifying that the integration between your frontend and backend services is functioning as expected.

Problem Statement 2:

1.Application Health Checker:

Please write a script that can check the uptime of an application and determine if it is functioning correctly or not. The script must accurately assess the application's status by checking HTTP status codes. It should be able to detect if the application is 'up', meaning it is functioning correctly, or 'down', indicating that it is unavailable or not responding.?

Answer

To create a test script using Selenium with Python and the `unittest` framework to check the uptime of an application by assessing HTTP status codes, you should use a combination of HTTP requests and Selenium. Selenium alone is not typically used to check HTTP status codes; however, you can use the `requests` library to perform this check.

Here’s a step-by-step guide to accomplish this:

1. **Set Up Your Environment**:

   Ensure you have the required packages installed:

   pip install selenium requests unittest

~~~
   ```
~~~

2. **Write the Test Script**:

   Below is a script that uses both `requests` to check the HTTP status code and `unittest` to run the test.

~~~
   ```python
   import unittest
   import requests
   from selenium import webdriver
   from selenium.webdriver.chrome.service import Service as ChromeService
   from webdriver_manager.chrome import ChromeDriverManager

   class TestApplicationUptime(unittest.TestCase):

       @classmethod
       def setUpClass(cls):
           # Initialize the WebDriver (using ChromeDriver in this example)
           cls.driver = webdriver.Chrome(service=ChromeService(ChromeDriverManager().install()))
           cls.url = 'http://your-application-url.com'  # Replace with your application's URL

       @classmethod
       def tearDownClass(cls):
           # Close the WebDriver instance
           cls.driver.quit()
~~~

```
    def check_http_status(self):
        try:
            response = requests.get(self.url)
            return response.status_code
        except requests.RequestException as e:
            self.fail(f'HTTP request failed: {e}')
            return None

    def test_application_is_up(self):
        status_code = self.check_http_status()
        self.assertEqual(status_code, 200, f'Application returned status code {status_code}, expected 200.')
        def test_application_functionality(self):
        # You can add a Selenium test to further check application functionality
        self.driver.get(self.url)
        # Example: Check if a specific element is present
        try:
            element = self.driver.find_element_by_id('some-element-id')  # Replace with an actual element ID
            self.assertTrue(element.is_displayed(), 'Expected element not found or not visible.')
        except Exception as e:
            self.fail(f'Selenium test failed: {e}')

if __name__ == '__main__':
    unittest.main()
```

3. **Explanation**:

- **`setUpClass`**: Initializes the WebDriver for Selenium and sets the URL of the application to test.

- **`tearDownClass`**: Quits the WebDriver after all tests are finished.

- **`check_http_status`**: Uses the `requests` library to perform an HTTP GET request to the application URL and returns the HTTP status code. If the request fails, it logs an error message.

- **`test_application_is_up`**: Checks if the HTTP status code is `200` (OK). If not, it fails the test with the actual status code received.

- **`test_application_functionality`**: Optionally, this Selenium-based test can check if specific functionality of the application works correctly. This can include checking for the presence of a specific element or other UI-related verifications.

4. **Run the Test**:

Save this script as `test_uptime.py` and run it using:

```
python test_uptime.py
```

5. **Adjust the Script**:

- **URL**: Replace `'http://your-application-url.com'` with the actual URL of your application.

- **Element Selector**: Update the `find_element_by_id` method to use the correct selector for your application’s functionality check.

- **Error Messages**: Customize the error messages to be more descriptive if needed.

This script will check if the application is up by verifying the HTTP status code and further assess its functionality using Selenium if desired.

----

2. System Health Monitoring Script:

Develop a script that monitors the health of a Linux system. It should check CPU usage, memory usage, disk space, and running processes. If any of these metrics exceed predefined thresholds (e.g., CPU usage > 80%), the script should send an alert to the console or a log file.?

Answer

Monitoring the health of a Linux system using Python in a unit test framework (such as `unittest`) can be accomplished by leveraging system utilities and libraries. Since `selenium` is typically used for web automation, it is not the appropriate tool for system monitoring. Instead, you can use Python’s built-in libraries and external packages to check system metrics.

Here’s a sample script using `psutil` for system monitoring in combination with the `unittest` framework. This script will:

1. Check CPU usage, memory usage, disk space, and running processes.

2. Send an alert if any of these metrics exceed predefined thresholds.

3. Log the results to the console or a file.

### Install Required Libraries

You will need the `psutil` library for system metrics. Install it using pip:

pip install psutil

### System Monitoring Script

Here’s a Python script using `unittest` and `psutil` to monitor system health:

```python
import unittest

import psutil

import logging


# Configure logging

logging.basicConfig(filename='system_health.log', level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
```

```
class TestSystemHealth(unittest.TestCase):
    def setUp(self):
        # Define thresholds
        self.cpu_threshold = 80.0
        self.memory_threshold = 80.0
        self.disk_threshold = 80.0

    def check_cpu_usage(self):
        cpu_usage = psutil.cpu_percent(interval=1)
        logging.info(f'CPU Usage: {cpu_usage}%')
        return cpu_usage

    def check_memory_usage(self):
        memory = psutil.virtual_memory()
        memory_usage = memory.percent
        logging.info(f'Memory Usage: {memory_usage}%')
        return memory_usage

    def check_disk_usage(self):
        disk = psutil.disk_usage('/')
        disk_usage = disk.percent
        logging.info(f'Disk Usage: {disk_usage}%')
        return disk_usage

    def check_running_processes(self):
        processes = [proc.name() for proc in psutil.process_iter()]
        logging.info(f'Running Processes: {processes}')
        return processes

    def test_cpu_usage(self):
        cpu_usage = self.check_cpu_usage()
        self.assertLessEqual(cpu_usage, self.cpu_threshold, f'CPU usage is too high: {cpu_usage}%')

    def test_memory_usage(self):
        memory_usage = self.check_memory_usage()
        self.assertLessEqual(memory_usage, self.memory_threshold, f'Memory usage is too high: {memory_usage}%')

    def test_disk_usage(self):
        disk_usage = self.check_disk_usage()
        self.assertLessEqual(disk_usage, self.disk_threshold, f'Disk usage is too high: {disk_usage}%')

    def test_running_processes(self):
        processes = self.check_running_processes()
        # You can add specific process checks here if needed
        self.assertIsNotNone(processes, 'No processes found.')

if __name__ == '__main__':
    unittest.main()
```

### Explanation

1. **Logging Configuration**: The script uses Python’s `logging` module to log system metrics to a file named `system_health.log`. This helps keep track of the metrics and alerts.

2. **Thresholds**: Thresholds for CPU, memory, and disk usage are defined in the `setUp` method.

3. **Metric Checks**:

   - `check_cpu_usage()`: Measures and logs CPU usage.

   - `check_memory_usage()`: Measures and logs memory usage.

   - `check_disk_usage()`: Measures and logs disk space usage.

   - `check_running_processes()`: Lists and logs running processes.

4. **Tests**:

   - `test_cpu_usage()`: Asserts that CPU usage is below the defined threshold.

   - `test_memory_usage()`: Asserts that memory usage is below the defined threshold.

   - `test_disk_usage()`: Asserts that disk usage is below the defined threshold.

   - `test_running_processes()`: Verifies that running processes are listed and non-empty.

5. **Running the Script**:

   Save the script as `test_system_health.py` and execute it using:

   ```bash
   python test_system_health.py
   ```

This script provides a basic framework for monitoring system health and can be extended to include more sophisticated checks or alerts based on your requirements.