**1. Introduction**

This document is designed to help to understand how to write test cases for GitHub Actions by walking through the modules, technologies, and strategies used in the provided test files. By following this guide, you'll be able to create your own test cases for different GitHub Actions and expand on the examples provided.

**2. Modules and Technologies Used**

**2.1. pytest**

* **Description**: pytest is a widely-used testing framework in Python that simplifies the process of writing and running tests. It supports fixtures, parameterized testing, and other advanced features, making it a robust tool for ensuring your code behaves as expected.
* **Naming Conventions**:
  + **Test File Names**: For pytest to automatically discover and run your tests, it expects test files to either start with test\_ or end with \_test.py. Examples include:
    - test\_example.py
    - example\_test.py
  + **Test Function Names**: Similarly, test functions within those files must start with test\_. For example:  def test\_example\_function():
  + assert True
  + Adhering to these naming conventions ensures that pytest can automatically find and execute your tests.
* **Usage in Test Files**:
  + pytest is the core framework used to structure and run all the test cases. For example, in the test case test\_send\_email\_notification\_with\_all\_vars, pytest is used to mock the smtp object calls and assert sending email notification behave as expected.  def test\_send\_email\_notification\_with\_all\_vars(monkeypatch):
  + # Use monkeypatch to set environment variables
  + monkeypatch.setenv('PROJECT\_GIT\_REPO', 'test-repo')
  + monkeypatch.setenv('NOTIFICATION\_MAP', '{"email\_recipients": ["test@example.com"], "subject": "Test Subject", "message": "Test Message"}')
  + monkeypatch.setenv('APP\_TYPE', 'web')
  + monkeypatch.setenv('BUILD\_URL', 'http://build-url.com')
  + monkeypatch.setenv('NOTIFY\_FLAGS', '{"send-teams-notification": true}')
  + # Define test data
  + message = "<p>This is a test message</p>"
  + recipients = ["test@example.com"]
  + email\_subject = "Test Subject"
  + # Mock the SMTP instance to prevent sending real emails
  + sent\_emails = []
  + def mock\_smtp(\*args, \*\*kwargs):
  + class MockSMTP:
  + def sendmail(self, from\_addr, to\_addrs, msg):
  + sent\_emails.append((from\_addr, to\_addrs, msg))
  + def quit(self):
  + pass
  + return MockSMTP()
  + monkeypatch.setattr(smtplib, 'SMTP', mock\_smtp)
  + # Call the function under test
  + send\_email\_notification(message, recipients, email\_subject)
  + # Assertions
  + assert len(sent\_emails) == 1
  + from\_addr, to\_addrs, msg = sent\_emails[0]
  + assert from\_addr == 'githubactions@kp.org'
  + assert to\_addrs == "test@example.com"
  + assert email\_subject in msg
  + assert message in msg

**2.2. unittest.mock**

* **Description**: The unittest.mock module is a powerful utility for creating mock objects during testing. It allows you to replace parts of your system under test and make assertions about how they were used.
* **Usage in Test Files**:
  + Mocking is essential in these test cases to simulate functions like subprocess.run and to verify that the correct parameters are passed. For example, in test\_generate\_test\_reports\_success, the subprocess.run function is mocked to prevent actual command execution during the test.  @mock.patch('main.subprocess.run', return\_value=mock.Mock())
  + @mock.patch('main.logging.info')
  + def test\_generate\_test\_reports\_success(mock\_logging\_info, mock\_subprocess\_run):
  + # Test implementation...

**2.3. monkey patch**

* **Description**: monkeypatch is a pytest fixture that allows you to modify objects and environment variables temporarily for testing purposes.
* **Usage in Test Files**:
  + monkeypatch is used extensively to set environment variables or modify functions for the duration of a test. For instance, in test\_set\_vars\_all\_values\_present, monkeypatch sets environment variables to simulate different test scenarios.  def test\_set\_vars\_all\_values\_present(monkeypatch):
  + config\_map = {
  + 'runtime\_version': '14.17.0',
  + 'args\_build': 'npm run build',
  + 'args\_test': 'npm test',
  + 'test\_flag\_enabled': 'true'
  + }
  + # Mocking os.system
  + mock\_os\_system = mock.Mock()
  + monkeypatch.setattr('main.os.system', mock\_os\_system)
  + set\_vars(config\_map)
  + # Assertions...

**2.4. Comparison: Mock and Monkeypatch**

**With Monkeypatch :**

def test\_send\_email\_notification\_no\_recipients(monkeypatch, caplog):

# Use monkey patch to set environment variables

monkeypatch.setenv('PROJECT\_GIT\_REPO', 'test-repo')

monkeypatch.setenv('NOTIFICATION\_MAP', '{}')

monkeypatch.setenv('APP\_TYPE', 'web')

monkeypatch.setenv('BUILD\_URL', 'http://build-url.com')

monkeypatch.setenv('NOTIFY\_FLAGS', '{"send-teams-notification": false}')

monkeypatch.setenv('LOG\_LEVEL', '20')

# Define test data

message = "<p>This is a test message</p>"

recipients = [] # No recipients

email\_subject = "Test Subject"

# Mock the SMTP instance to prevent sending real emails

sent\_emails = []

def mock\_smtp(\*args, \*\*kwargs):

class MockSMTP:

def sendmail(self, from\_addr, to\_addrs, msg):

sent\_emails.append((from\_addr, to\_addrs, msg))

def quit(self):

pass

return MockSMTP()

monkeypatch.setattr(smtplib, 'SMTP', mock\_smtp)

# Call the function under test with logging capture

with caplog.at\_level(logging.INFO):

send\_email\_notification(message, recipients, email\_subject)

# Assertions

assert let(sent\_emails) == 0 # Ensure no emails were sent

assert "No emails addresses configured." in caplog.text

**With Mock :**

def test\_send\_email\_notification\_no\_recipients(mock\_smtp):

# Define test data

message = "<p>This is a test message</p>"

recipients = [] # No recipients

email\_subject = "Test Subject"

# Mock the SMTP instance to prevent sending real emails

mock\_smtp\_instance = mock\_smtp.return\_value

mock\_smtp\_instance.sendmail = MagicMock()

# Call the function under test

send\_email\_notification(message, recipients, email\_subject)

# Assertions

mock\_smtp.assert\_not\_called() # No SMTP actions should be performed

mock\_smtp\_instance.quit.assert\_not\_called() # No SMTP actions should be performed

**2.5. Function Mocking Using monkeypatch**

* **Description**: In addition to mocking attributes or environment variables, monkeypatch can also be used to mock entire functions within your code, which is especially useful when you want to avoid side effects such as sending emails or making HTTP requests during tests.
* **Usage Example**:

**Mocking smtplib.SMTP in send\_email\_notification:**

def test\_send\_email\_notification\_with\_mock(monkeypatch):

# Mock SMTP class

class MockSMTP:

def \_\_init\_\_(self, host, port):

self.host = host

self.port = port

self.sent\_messages = []

def sendmail(self, from\_addr, to\_addrs, msg):

self.sent\_messages.append((from\_addr, to\_addrs, msg))

def quit(self):

pass

# Replace smtplib.SMTP with the mock class

monkeypatch.setattr(smtplib, 'SMTP', MockSMTP)

# Prepare test data

message = "<p>This is a test message</p>"

recipients = ["test@example.com"]

email\_subject = "Test Subject"

# Call the function under test

send\_email\_notification(message, recipients, email\_subject)

# Assert the expected behavior

mock\_smtp = smtplib.SMTP('mta.kp.org', 25)

assert len(mock\_smtp.sent\_messages) == 1 # One email should be sent

assert mock\_smtp.sent\_messages[0][0] == 'githubactions@kp.org' # From address

assert mock\_smtp.sent\_messages[0][1] == "test@example.com" # To address

assert "Test Subject" in mock\_smtp.sent\_messages[0][2] # Email subject in the message

assert "<p>This is a test message</p>" in mock\_smtp.sent\_messages[0][2] # Email body

**Mocking notification\_message in send\_environment\_notification:**

def test\_send\_environment\_notification\_with\_mock(monkeypatch):

# Mock notification\_message function

def mock\_notification\_message(message, teams\_channel, job\_status):

return f"Mocked: {message}, {teams\_channel}, {job\_status}"

# Replace the original notification\_message with the mock

monkeypatch.setattr('main.notification\_message', mock\_notification\_message)

# Prepare test data

notification\_map = {

'environment': 'production',

'artifact\_name': 'v1.2.3',

'message': 'Deployment successful!'

}

job\_status = 'Success'

# Call the function under test

result = send\_environment\_notification(notification\_map, job\_status)

# Assert the expected behavior (in this case, the mock return value)

assert result == "Mocked: Environment: <b>production</b>, Application Type: <b>{app\_type}</b>, Artifact Version : <b>v1.2.3, Workflow status : <b>Success</b>, Deployment successful!, teams-channel-id-prod, Success"

**2.6. yaml**

* **Description**: The yaml module provides the ability to parse and produce YAML data. It's particularly useful for configuration files.
* **Usage in Test Files**:
  + The yaml.safe\_load method is used to parse environment variables containing YAML-formatted data, which is then used in the tests for functions like send\_environment\_notification.  env\_notification\_map = yaml.safe\_load(os.getenv('ENV\_NOTIFICATION\_MAP'))
  + teams\_channel = env\_notification\_map.get(app\_type).get(deploy\_env)

**3. Writing Test Cases**

**3.1. Structure of a Test Case**

Each test case generally follows this structure:

1. **Setup**: Initialize the test environment using tools like monkeypatch or mock.
2. **Execution**: Call the function under test with the appropriate parameters.
3. **Verification**: Use assertions to verify the expected outcomes.

**3.2. Example Test Case Breakdown**

**Example: test\_generate\_test\_reports\_success** This test case verifies that the generate\_test\_reports function behaves as expected when all necessary inputs are provided.

@mock.patch('main.subprocess.run', return\_value=mock.Mock())

@mock.patch('main.logging.info')

def test\_generate\_test\_reports\_success(mock\_logging\_info, mock\_subprocess\_run):

build\_var\_map = {

'args\_test': 'npm test',

'build\_group': {

'js-lcov-report-path': '/path/to/lcov-report',

'cobertura': True,

'html-reports': {

'pipeline-coverage-report': {

'report-dir': 'coverage-report-dir'

}

}

}

}

with mock.patch('main.workspace', '/fake/workspace'):

generate\_test\_reports(build\_var\_map)

mock\_logging\_info.assert\_any\_call('Generating test report using npm test')

mock\_subprocess\_run.assert\_any\_call('export LOG\_LEVEL=ERROR && npm test', shell=True, check=True, timeout=3600)

**3.3. Handling Different Scenarios**

* **Subprocess Failures**: Tests like test\_generate\_test\_reports\_subprocess\_failure demonstrate how to handle subprocess errors by mocking exceptions like subprocess.CalledProcessError. pythonCopy code  @mock.patch('main.subprocess.run', side\_effect=subprocess.CalledProcessError(1, 'npm test'))
* def test\_generate\_test\_reports\_subprocess\_failure(mock\_run, caplog):
* build\_var\_map = {'args\_test': 'npm test', 'build\_group': {}}
* with mock.patch('main.workspace', '/fake/workspace'):
* generate\_test\_reports(build\_var\_map)
* assert any("Command 'npm test' returned non-zero exit status 1." in record.message for record in caplog.records)
* **No Arguments Provided**: Tests such as test\_generate\_test\_reports\_no\_args\_test show how to handle scenarios where required parameters are missing, ensuring the function behaves gracefully. pythonCopy code  def test\_generate\_test\_reports\_no\_args\_test(monkeypatch):
* build\_var\_map = {'build\_group': {}}
* mock\_logging\_info = mock.Mock()
* monkeypatch.setattr('main.logging.info', mock\_logging\_info)
* generate\_test\_reports(build\_var\_map)
* mock\_logging\_info.assert\_not\_called()
* **Notification Testing**: Functions like send\_email\_notification and notification\_message are tested to ensure notifications are sent correctly, with various scenarios such as no recipients being covered in test\_send\_email\_notification\_no\_recipients. pythonCopy code  def test\_send\_email\_notification\_no\_recipients(monkeypatch):
* monkeypatch.setenv('PROJECT\_GIT\_REPO', 'test-repo')
* monkeypatch.setenv('NOTIFICATION\_MAP', '{}')
* message = "<p>This is a test message</p>"
* recipients = [] # No recipients
* email\_subject = "Test Subject"
* with patch('main.smtplib.SMTP') as mock\_smtp:
* send\_email\_notification(message, recipients, email\_subject)
* mock\_smtp.assert\_not\_called() # No SMTP actions should be performed

**4. Best Practices**

* **Isolation**: Ensure each test is independent and does not rely on external systems by using mocks and fixtures like monkeypatch.
* **Comprehensive Coverage**: Cover both positive and negative scenarios, including edge cases like timeouts, failures, and missing configurations.
* **Readable and Maintainable**: Write clear and descriptive test case names that explain the scenario being tested, and include comments where necessary.