SKILLS | DevOps and Cloud Computing

Integration Testing, TDD/BDD, and Code Quality Analysis

Objective

- Understand integration testing and explore popular frameworks like Selenium and Cypress.
- Learn the principles of Test-Driven Development (TDD) an Behavior-Driven Development (BDD).
- Utilize tools for code coverage and quality analysis.
- Apply integration testing and TDD/BDD methodologies in real-world scenarios.

PW SKILLS

Integration testing and how it differs from unit testing

Integration Testing

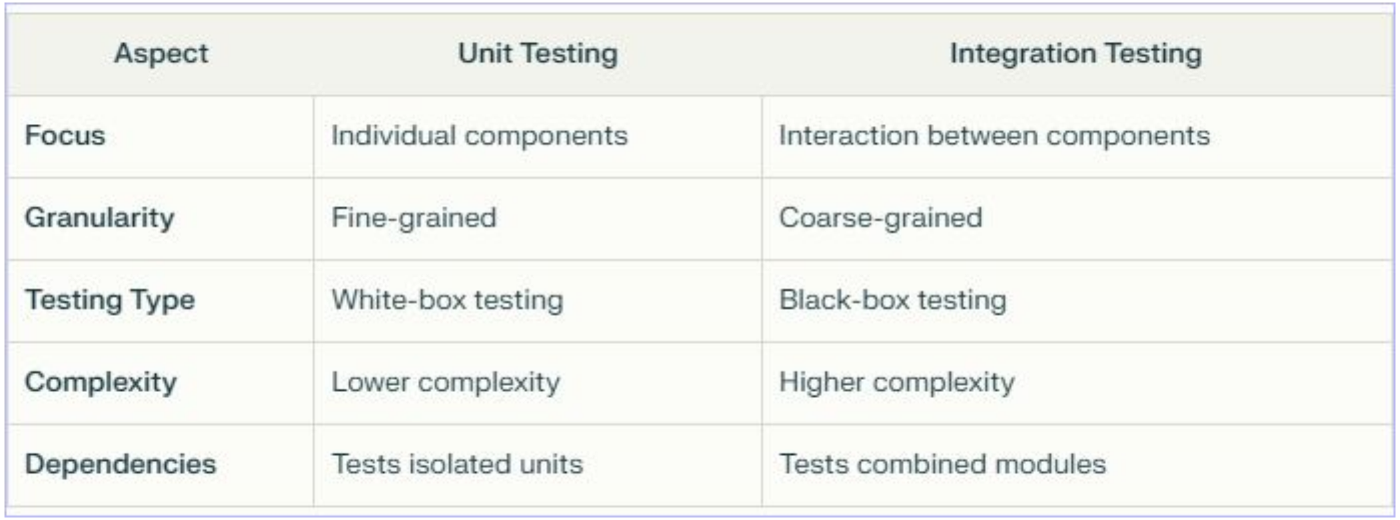
Integration Testing

Definition: Integration testing examines how multiple software modules or components interact when combined. It aims to identify issues in the interfaces and interactions between these components, such as data flow, communication, or integration with external systems like databases or APIs.

Purpose: To ensure that integrated parts of the application work together as expected, even if they were developed independently

Integration Testing

Key Differences



Aspect	Unit Testing	Integration Testing
Focus	Individual components	Interaction between components
Granularity	Fine-grained	Coarse-grained
Testing Type	White-box testing	Black-box testing
Complexity	Lower complexity	Higher complexity
Dependencies	Tests isolated units	Tests combined modules

Introduce popular frameworks

Frameworks

1. **Selenium:**
   - **Purpose:** Automates browser testing for web applications.
   - **Features:** Open-source, supports multiple programming languages (e.g., Java, Python), and enables cross-browser testing using tools like Selenium WebDriver and Selenium Grid. It is widely used for functional and regression testing.

2. **Cypress:**
    - **Purpose:** A modern framework for end-to-end testing of web applications.
    - **Features:** Focuses on developer-friendly testing with real-time reloading, debugging, and fast execution. It is particularly effective for front-end testing with built-in support for JavaScript frameworks.

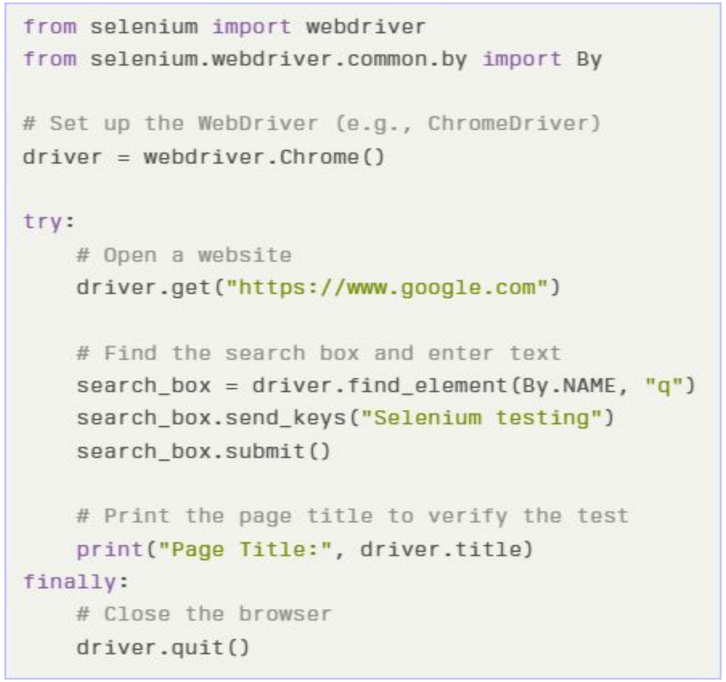
3. **Postman/Newman:**
    - **Purpose:** API testing and integration.
    - Features: Postman provides a user-friendly interface for designing, testing, and automating APIs. Newman is its command-line companion for running Postman collections in CI/CD pipelines. Both are essential for validating API functionality, performance, and security.

PW SKILLS

Demonstrating running a basic Selenium or Cypress test

Let’s do it

Running a Basic Selenium Test (Python Example)



```python
from selenium import webdriver
from selenium.webdriver.common.by import By

# Set up the WebDriver (e.g., ChromeDriver)
driver = webdriver.Chrome()

try:
    # Open a website
    driver.get("https://www.google.com")

    # Find the search box and enter text
    search_box = driver.find_element(By.NAME, "q")
    search_box.send_keys("Selenium testing")
    search_box.submit()

    # Print the page title to verify the test
    print("Page Title:", driver.title)
finally:
    # Close the browser
    driver.quit()
```

Running a Basic Cypress Test

- **Install Cypress:** Run ‘npm install cypress’ in your project directory.
- **Write Test Code (‘cypress/e2e/test.cy.js’):**

```
describe('Google Search Test', () => {
    it('Searches for Selenium testing', () => {
        // Visit Google
        cy.visit('https://www.google.com');

        // Find the search box, type a query, and submit
        cy.get('input[name="q"]').type('Selenium testing{enter}');

        // Verify the page title contains the query
        cy.title().should('include', 'Selenium testing');
    });
});
```

Run Test: Execute ‘npx cypress open’ to run the test in Cypress Test Runner.

Pop Quiz

Q. What is the purpose of Selenium in CI pipelines??

A

To manage source code repositories

B

To automate browser-based testing

Pop Quiz

Q. What is the purpose of Selenium in CI pipelines??

A

To manage source code repositories

B

To automate browser-based testing

Explaining the TDD process

TDD process

The Test-Driven Development (TDD) process follows an iterative cycle known as Red-Green-Refactor:

1. **Write a Failing Test (Red):**

- Create a test case for the desired feature or functionality.
- The test will fail initially because the feature has not yet been implemented, ensuring the test's validity

TDD process

2. **Implement Code (Green):**

- Write the minimal code required to make the failing test pass.
- Focus on functionality rather than code quality at this stage.

3. **Refactor Code (Refactor):**

- Improve the code's structure, readability, and maintainability without altering its behavior.
- Ensure all tests still pass after refactoring.

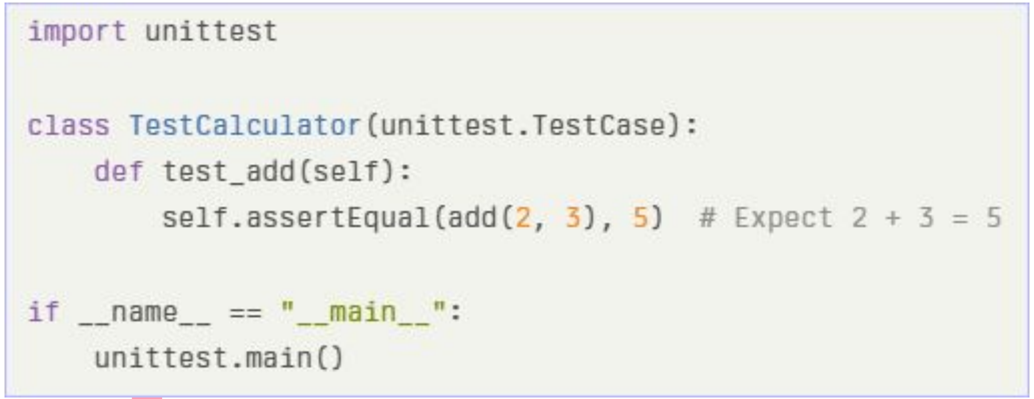
PW SKILLS

Demonstrating writing a simple test-first approach using TDD

Let’s do it

Demonstrating TDD with a Simple Example: Calculator's ‘add’ Function

Step 1: Write a Failing Test (Red)
Create a test to check if the ‘add’ function correctly sums two numbers. This test will fail initially since the function doesn't exist yet.



```python
import unittest

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)  # Expect 2 + 3 = 5

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

Let’s do it

Step 2: Write Minimal Code to Pass the Test (Green)
Implement the ‘add’ function with just enough code to make the test pass.

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

Step 3: Refactor the Code (Refactor)
Clean up the code if necessary. In this case, no refactoring is needed as the implementation is already simple and efficient.

PW SKILLS

Introducing BDD and discuss its advantages in collaboration and clarity

BDD & it’s advantages

Behavior-Driven Development (BDD) is an Agile software development methodology that focuses on defining the behavior of an application in collaboration with stakeholders, using natural language and concrete examples.

- It extends **Test-Driven Development (TDD)** by emphasizing communication and shared understanding between developers, testers, and business stakeholders.

Advantages of BDD:

1. Improved Collaboration
2. Enhanced Clarity
3. Early Defect Detection
4. Focus on User Needs
5. Streamlined Development

SKILLS

Discuss Gherkin syntax in BDD frameworks (Cucumber, Behave, SpecFlow)

Gherkin syntax

Gherkin is a simple, human-readable language used in BDD frameworks like Cucumber, Behave, and SpecFlow to define test scenarios. It bridges the gap between technical and non-technical stakeholders by describing software behavior in plain English.

Key Features of Gherkin Syntax:

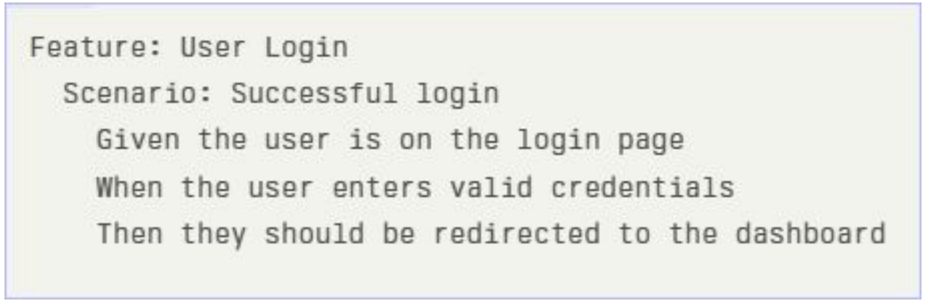
1. **Structure:**

Written in .feature files with a clear structure:

- Feature: Describes the functionality being tested.
- Scenario: Represents a specific behavior or use case.
- Steps: Defined using keywords like Given, When, Then, And, and But.

Gherkin syntax

2. **Example Syntax:**

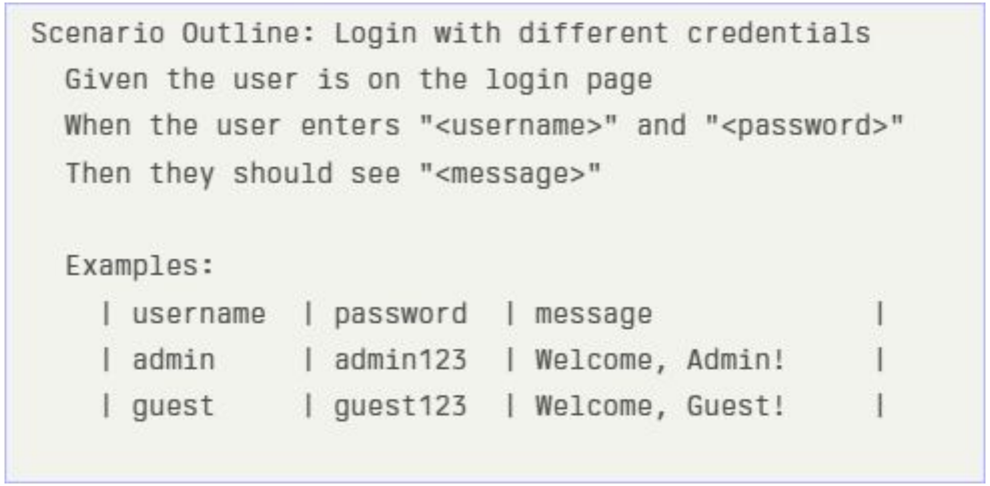


```
Feature: User Login
  Scenario: Successful login
    Given the user is on the login page
    When the user enters valid credentials
    Then they should be redirected to the dashboard
```

Gherkin syntax

3. Scenario Outline:

- Used for parameterized tests with multiple inputs.



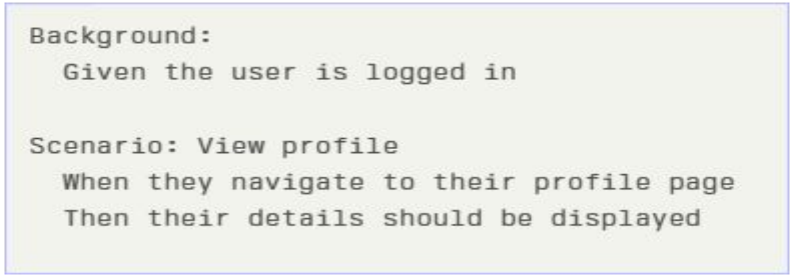
```
Scenario Outline: Login with different credentials
  Given the user is on the login page
  When the user enters "<username>" and "<password>"
  Then they should see "<message>"

  Examples:
    | username  | password  | message              |
    | admin     | admin123  | Welcome, Admin!      |
    | guest     | guest123  | Welcome, Guest!      |
```

Gherkin syntax

4. Background:

- Defines common preconditions for multiple scenarios.



```
Background:
  Given the user is logged in

Scenario: View profile
  When they navigate to their profile page
  Then their details should be displayed
```

Framework-Specific Features

- **Cucumber:** Supports multiple languages (e.g., Java, Python) and integrates with various tools for automation.
- **Behave:** Python-based BDD framework ideal for web and non-web applications.
- SpecFlow: Designed for .NET projects with seamless integration into Visual Studio.

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Why code coverage is essential for maintaining high-quality software

Code coverage

Code coverage is essential for maintaining high-quality software because it ensures thorough testing, improves code reliability, and facilitates better development practices. Here are the key reasons:

- Identifies Untested Code
- Improves Code Quality
- Facilitates Refactoring
- Enhances Reliability
- Supports Risk Mitigation
- Boosts Developer Confidence

PW SKILLS

Introducing popular tools for different ecosystems

Popular Tools

Here is an introduction to popular code coverage tools for Python, Java, and JavaScript ecosystems:

pytest-cov (Python): This is a plugin for the ‘pytest’ testing framework that integrates with ‘coverage.py’ to generate detailed code coverage reports.

- It supports multiple formats like terminal output, HTML, XML, and JSON. It is widely used due to its simplicity, ability to handle subprocesses, and compatibility with distributed testing setups.

Popular Tools

JaCoCo (Java): JaCoCo is a robust Java code coverage library often integrated with build tools like Maven and Gradle. It provides metrics for line, branch, and instruction coverage and generates reports in formats such as HTML, XML, and CSV.

- It is commonly used in CI/CD pipelines to enforce coverage thresholds.

Istanbul/nyc (JavaScript): Istanbul is a popular JavaScript code coverage tool that works with many test runners. Its command-line interface, nyc, simplifies integration into testing workflows.

- It supports multiple report formats (e.g., HTML, LCOV) and can measure statement, branch, line, and function coverage

The limitations of 100% code coverage and the importance of meaningful tests

Let’s see

Limitations of 100% Code Coverage:

1. Doesn't Guarantee Bug-Free Code
2. Expensive and Time-Consuming
3. Diminishing Returns
4. False Sense of Security
5. Impractical for Certain Code

Importance of Meaningful Tests:

1. Focus on Critical Paths
2. Quality Over Quantity
3. Improved Maintainability
4. Risk Mitigation

PW SKILLS

Demonstrating running a coverage report on a test suite

Let’s do it

To run a coverage report on a test suite, follow these steps for the pytest-cov tool in Python:

Steps:

1. **Install Required Tools:**

Install pytest and pytest-cov using pip:

```
pip install pytest pytest-cov
```

2. **Run Tests with Coverage:**

Execute your test suite with the ‘--cov' option to generate a coverage report:

```
pytest --cov=<source_directory>
```

Replace <source directory> with the directory containing your source code.

3. View Coverage Report:

- The coverage summary will be displayed in the terminal, showing the percentage of code covered.
- To generate an HTML report, add the --cov-report=html option:

```
pytest --cov=<source_directory> --cov-report=html
```

- This creates an ‘htmlcov’ directory containing detailed coverage results.

4. Analyze Results:

Open the ‘index.html’ file in the ‘htmlcov’ folder to view line-by-line coverage visualization in your browser.

Discuss how TDD/BDD fits into CI/CD workflows

Let’s discuss

How TDD/BDD Fits into CI/CD:

1. Early Bug Detection
2. Automated Testing
3. Collaboration and Clarity
4. Continuous Validation
5. Improved Release Confidence

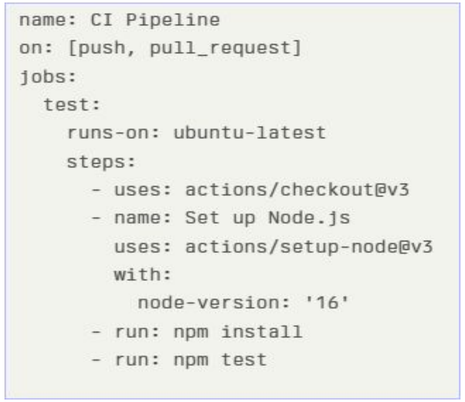
Automated testing in pipelines using GitHub Actions, Jenkins, or GitLab CI

Automated testing

1. **GitHub Actions**

- **Setup:** Define workflows in YAML files under the '.github/workflows/ directory'.
- **Trigger:** Tests can be triggered by events like pushes, pull requests, or scheduled runs.

Example Workflow:



```yaml
name: CI Pipeline
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '16'
      - run: npm install
      - run: npm test
```

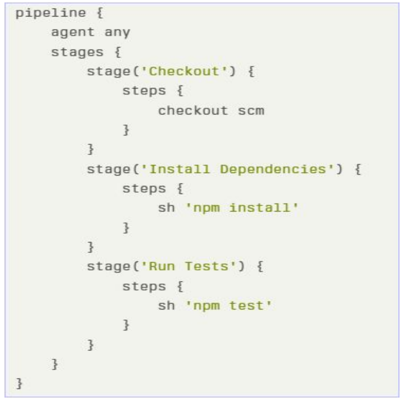
- This workflow installs dependencies and runs tests automatically

Automated testing

2. Jenkins

- **Setup:** Use Jenkins pipelines defined in a 'Jenkinsfile'.
- **Trigger:** Pipelines can be triggered by webhooks, cron schedules, or manual triggers.

Example Pipeline:



```
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }
        stage('Install Dependencies') {
            steps {
                sh 'npm install'
            }
        }
        stage('Run Tests') {
            steps {
                sh 'npm test'
            }
        }
    }
}
```

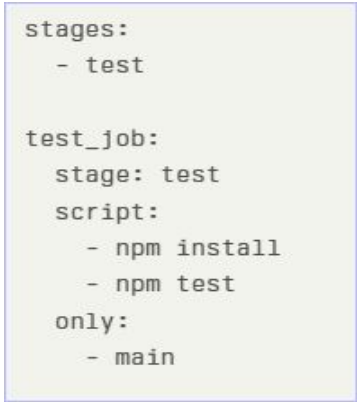
- Jenkins provides detailed logs and integrates with many plugins for extended functionality

Automated testing

3. GitLab CI

- **Setup:** Define pipelines in a ‘.gitlab-ci.yml’ file.
- **Trigger:** Pipelines run on pushes, merges, or scheduled events.

Example Pipeline:



```
stages:
  - test

test_job:
  stage: test
  script:
    - npm install
    - npm test
  only:
    - main
```

- GitLab provides built-in CI/CD capabilities with easy integration into GitLab repositories.

Demonstrating setting up an integration test step in a CI pipeline

Let’s do it

Here’s how to set up an integration test step in a CI pipeline using GitHub Actions:

Example Workflow for Integration Tests:

```
name: Integration Tests

on: [push, pull request]

jobs:
  integration-test:
    runs-on: ubuntu-latest
    services:
      db:
        image: postgres:13
        ports:
          - 5432:5432
        env:
          POSTGRES_USER: testuser
          POSTGRES_PASSWORD: testpassword
          POSTGRES_DB: testdb
        options: >-
          --health-cmd="pgisready -U testuser"
          --health-interval=10
```

```
--health-timeout=5s
          --health-retries=3

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Node.js (example)
        uses: actions/setup-node@v3
        with:
          node-version: '16'

      - name: Install dependencies
        run: npm install

      - name: Run integration tests
        env:
          DATABASE URL:
postgres://test_user:testpassword@localhost:5432/testdb
        run: npm run test:integration
```

PW SKILLS

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this session
- Review your lecture notes
- Note down questions and queries regarding this session and consult the teaching assistants

Thanks

!