Hybrid Framework

# Hybrid Framework in PyTest: A Comprehensive Guide

A **Hybrid Testing Framework** combines the strengths of multiple testing approaches, such as **Data-Driven** and **Keyword-Driven** testing, to create a robust, reusable, and scalable structure. In PyTest, this can be achieved by leveraging its powerful plugins, fixtures, and parameterization capabilities, alongside external tools for configuration and reporting.

**Core Components of a Hybrid Framework**

1. **Test Data Management**:
   * Use external sources (CSV, JSON, Excel, or databases) to store test data.
   * Enable **Data-Driven Testing** by parameterizing tests with data sets.
2. **Keyword Implementation**:
   * Define reusable functions or keywords that map to specific actions (e.g., login, navigation).
   * These keywords can be invoked in test scripts for modularity.
3. **Configuration Management**:
   * Centralize configuration files (e.g., config.yaml) for environment settings like URLs, database credentials, etc.
4. **Test Structure**:
   * Maintain a clear directory structure for better organization:
   * project/
   * ├── tests/
   * │ ├── test\_cases/
   * │ ├── test\_data/
   * │ ├── test\_keywords.py
   * │ └── conftest.py
   * ├── utils/
   * │ ├── data\_reader.py
   * │ └── custom\_logger.py
   * ├── config/
   * │ └── config.yaml
   * ├── reports/
   * └── requirements.txt
5. **Fixtures**:
   * Use **PyTest fixtures** for setup and teardown operations.
   * Fixtures allow sharing state across tests and managing preconditions/postconditions.
6. **Reporting**:
   * Integrate PyTest plugins like **pytest-html** or **pytest-allure** for detailed reporting.

**Implementation Steps**

**1. Test Data Management**

Example: Using a CSV file for login credentials.

**test\_data/login\_data.csv**

username,password,expected

user1,pass1,success

user2,pass2,failure

**utils/data\_reader.py**

import csv

def read\_csv(file\_path):

with open(file\_path, mode="r") as file:

reader = csv.DictReader(file)

return [row for row in reader]

**2. Keyword Implementation**

Define reusable actions.

**tests/test\_keywords.py**

from selenium import webdriver

def open\_browser():

driver = webdriver.Chrome()

return driver

def login(driver, username, password):

driver.find\_element\_by\_id("username").send\_keys(username)

driver.find\_element\_by\_id("password").send\_keys(password)

driver.find\_element\_by\_id("login-button").click()

def close\_browser(driver):

driver.quit()

**3. Test Configuration**

Centralize configurations using YAML.

**config/config.yaml**

base\_url: "https://example.com"

browser: "chrome"

**utils/config\_loader.py**

import yaml

def load\_config(file\_path):

with open(file\_path, "r") as file:

return yaml.safe\_load(file)

**4. Fixtures**

Manage setup and teardown with fixtures.

**tests/conftest.py**

import pytest

from tests.test\_keywords import open\_browser, close\_browser

from utils.config\_loader import load\_config

@pytest.fixture(scope="session")

def config():

return load\_config("config/config.yaml")

@pytest.fixture(scope="function")

def browser(config):

driver = open\_browser()

driver.get(config["base\_url"])

yield driver

close\_browser(driver)

**5. Parameterization**

Run tests with multiple data sets.

**tests/test\_cases/test\_login.py**

import pytest

from utils.data\_reader import read\_csv

from tests.test\_keywords import login

@pytest.mark.parametrize("test\_data", read\_csv("test\_data/login\_data.csv"))

def test\_login\_functionality(browser, test\_data):

username = test\_data["username"]

password = test\_data["password"]

expected = test\_data["expected"]

login(browser, username, password)

# Validate result (pseudo-code)

if expected == "success":

assert "Dashboard" in browser.title

else:

assert "Login failed" in browser.page\_source

**6. Reporting**

Install **pytest-html** for generating HTML reports:

pip install pytest-html

Run tests with reporting:

pytest --html=reports/test\_report.html --self-contained-html

**Advantages of a Hybrid Framework in PyTest**

1. **Modularity**:
   * Keywords and test data are decoupled, enabling reusable components.
2. **Scalability**:
   * Easy to add new tests or extend existing ones.
3. **Flexibility**:
   * Combines multiple approaches for specific test needs.
4. **Readability**:
   * Clear separation of data, keywords, and logic improves maintainability.

**Best Practices**

1. **Error Handling**:
   * Implement error handling in keywords for robust scripts.
2. **Version Control**:
   * Use Git for tracking changes to test cases and configurations.
3. **CI/CD Integration**:
   * Automate test execution using tools like Jenkins or GitHub Actions.
4. **Documentation**:
   * Maintain clear documentation for keywords, test cases, and framework usage.

# Hybrid Framework in TestNG: A Comprehensive Guide

A **Hybrid Framework** in TestNG combines the strengths of **Data-Driven** and **Keyword-Driven** testing, making it flexible, reusable, and scalable. TestNG provides robust features like annotations, parallel execution, and integration with external tools (Excel, databases, etc.) to implement a hybrid framework effectively.

**Core Components of a Hybrid Framework in TestNG**

1. **Test Data Management**:
   * Use external files (e.g., Excel, CSV) or databases to manage test data.
   * Implement **Data-Driven Testing** using the @DataProvider annotation.
2. **Keyword Implementation**:
   * Create reusable methods (keywords) for specific actions (e.g., login, navigation).
   * Use these methods across test scripts to ensure modularity.
3. **Configuration Management**:
   * Use a centralized configuration file (e.g., config.properties) to manage settings like URLs, browser types, and credentials.
4. **Test Structure**:
   * Organize the framework for maintainability:
   * project/
   * ├── src/
   * │ ├── main/
   * │ │ ├── config/
   * │ │ │ └── config.properties
   * │ │ ├── keywords/
   * │ │ │ └── ActionKeywords.java
   * │ │ ├── utils/
   * │ │ │ ├── ExcelUtils.java
   * │ │ │ └── ConfigReader.java
   * │ └── test/
   * │ ├── test\_cases/
   * │ └── testng.xml
   * └── reports/
5. **Annotations**:
   * Leverage TestNG annotations (@BeforeSuite, @Test, @DataProvider) for structured execution.
6. **Reporting**:
   * Utilize TestNG’s built-in reports or integrate with **ExtentReports** for detailed reports.

**Implementation Steps**

**1. Test Data Management**

Example: Using an Excel file for login data.

**Excel File (TestData.xlsx)**

| **Username** | **Password** | **Expected Result** |
| --- | --- | --- |
| user1 | pass1 | success |
| user2 | pass2 | failure |

**Excel Utility Class**:

import org.apache.poi.ss.usermodel.\*;

import java.io.FileInputStream;

public class ExcelUtils {

private static Workbook workbook;

public static void loadExcel(String filePath) throws Exception {

FileInputStream fis = new FileInputStream(filePath);

workbook = WorkbookFactory.create(fis);

}

public static String getCellData(String sheetName, int row, int column) {

Sheet sheet = workbook.getSheet(sheetName);

Row dataRow = sheet.getRow(row);

Cell cell = dataRow.getCell(column);

return cell.toString();

}

public static int getRowCount(String sheetName) {

return workbook.getSheet(sheetName).getLastRowNum();

}

}

**2. Keyword Implementation**

Create a class for reusable actions.

**ActionKeywords.java**:

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

public class ActionKeywords {

WebDriver driver;

public ActionKeywords(WebDriver driver) {

this.driver = driver;

}

public void openBrowser(String url) {

driver.get(url);

}

public void inputText(WebElement element, String text) {

element.sendKeys(text);

}

public void clickElement(WebElement element) {

element.click();

}

public void closeBrowser() {

driver.quit();

}

}

**3. Configuration Management**

Centralize configurations using a properties file.

**config.properties**:

baseUrl=https://example.com

browser=chrome

**ConfigReader.java**:

import java.io.FileInputStream;

import java.util.Properties;

public class ConfigReader {

private static Properties properties;

public static void loadConfig(String filePath) throws Exception {

FileInputStream fis = new FileInputStream(filePath);

properties = new Properties();

properties.load(fis);

}

public static String getProperty(String key) {

return properties.getProperty(key);

}

}

**4. Parameterization with DataProvider**

Use @DataProvider for reading data from Excel.

**LoginTest.java**:

import org.testng.annotations.\*;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.By;

public class LoginTest {

WebDriver driver;

ActionKeywords actions;

@BeforeSuite

public void setup() throws Exception {

ConfigReader.loadConfig("src/main/config/config.properties");

driver = WebDriverManager.chromedriver().setup();

actions = new ActionKeywords(driver);

}

@DataProvider(name = "LoginData")

public Object[][] getLoginData() throws Exception {

ExcelUtils.loadExcel("src/main/test\_data/TestData.xlsx");

int rows = ExcelUtils.getRowCount("LoginSheet");

Object[][] data = new Object[rows][3];

for (int i = 0; i < rows; i++) {

data[i][0] = ExcelUtils.getCellData("LoginSheet", i, 0);

data[i][1] = ExcelUtils.getCellData("LoginSheet", i, 1);

data[i][2] = ExcelUtils.getCellData("LoginSheet", i, 2);

}

return data;

}

@Test(dataProvider = "LoginData")

public void testLogin(String username, String password, String expected) {

actions.openBrowser(ConfigReader.getProperty("baseUrl"));

WebElement userField = driver.findElement(By.id("username"));

WebElement passField = driver.findElement(By.id("password"));

WebElement loginButton = driver.findElement(By.id("login-button"));

actions.inputText(userField, username);

actions.inputText(passField, password);

actions.clickElement(loginButton);

String actualTitle = driver.getTitle();

if (expected.equals("success")) {

assert actualTitle.contains("Dashboard");

} else {

WebElement errorMessage = driver.findElement(By.id("error-message"));

assert errorMessage.isDisplayed();

}

actions.closeBrowser();

}

}

**5. TestNG XML Configuration**

Control test execution through an XML file.

**testng.xml**:

<suite name="Hybrid Framework Suite">

<test name="Login Tests">

<classes>

<class name="test\_cases.LoginTest" />

</classes>

</test>

</suite>

Run tests with:

mvn test

**6. Reporting**

Integrate **ExtentReports** for advanced reporting.

1. Add ExtentReports dependency in pom.xml:

<dependency>

<groupId>com.aventstack</groupId>

<artifactId>extentreports</artifactId>

<version>5.0.9</version>

</dependency>

1. Generate reports in your tests:

import com.aventstack.extentreports.\*;

public class ReportManager {

private static ExtentReports extent;

private static ExtentTest test;

public static void initReport() {

ExtentSparkReporter spark = new ExtentSparkReporter("reports/ExtentReport.html");

extent = new ExtentReports();

extent.attachReporter(spark);

}

public static void logTest(String name) {

test = extent.createTest(name);

}

public static void flushReport() {

extent.flush();

}

}

**Advantages of a Hybrid Framework in TestNG**

1. **Modularity**: Clear separation of test logic, data, and configuration.
2. **Scalability**: Easy to add new tests or extend functionality.
3. **Flexibility**: Combines the best features of multiple frameworks.
4. **Reusability**: Reusable keywords and test data improve efficiency.
5. **Reporting**: Advanced reporting options make debugging easier.

# Hybrid Framework in Cypress: A Comprehensive Guide

A **Hybrid Framework** in Cypress blends **Data-Driven Testing**, **Keyword-Driven Testing**, and Cypress's core features, like JavaScript syntax, plugins, and command chaining. It provides a robust structure for end-to-end testing by combining reusability, scalability, and flexibility.

**Core Components of a Hybrid Framework in Cypress**

1. **Test Data Management**:
   * Use external files (e.g., JSON, Excel, CSV) or fixtures for managing test data.
   * Enable **Data-Driven Testing** by parameterizing tests with dynamic datasets.
2. **Keyword Implementation**:
   * Define reusable custom commands in Cypress (via commands.js) for modularity.
   * These commands act as **keywords**, which can be reused across tests.
3. **Configuration Management**:
   * Manage environment-specific settings like base URLs and API keys in cypress.config.js or environment files (cypress.env.json).
4. **Test Structure**:
   * Organize the project for maintainability and scalability:
   * project/
   * ├── cypress/
   * │ ├── fixtures/
   * │ │ └── test\_data.json
   * │ ├── integration/
   * │ │ ├── test\_cases/
   * │ │ │ └── login.spec.js
   * │ │ ├── pages/
   * │ │ │ └── LoginPage.js
   * │ │ ├── keywords/
   * │ │ │ └── commonActions.js
   * │ ├── plugins/
   * │ │ └── index.js
   * │ ├── support/
   * │ │ ├── commands.js
   * │ │ └── index.js
   * ├── cypress.config.js
   * ├── cypress.env.json
   * └── reports/
5. **Reporting**:
   * Integrate Cypress plugins for advanced reporting, such as **Mochawesome**.

**Implementation Steps**

**1. Test Data Management**

**cypress/fixtures/test\_data.json**

{

"loginData": [

{ "username": "user1", "password": "pass1", "expected": "success" },

{ "username": "user2", "password": "pass2", "expected": "failure" }

]

}

Access data in tests:

cy.fixture("test\_data").then((data) => {

const loginData = data.loginData;

});

**2. Keyword Implementation**

Define reusable custom commands.

**cypress/support/commands.js**

Cypress.Commands.add("login", (username, password) => {

cy.get("#username").type(username);

cy.get("#password").type(password);

cy.get("#login-button").click();

});

Cypress.Commands.add("verifyTitle", (expectedTitle) => {

cy.title().should("include", expectedTitle);

});

**3. Page Object Model**

Use the Page Object Model for modularity.

**cypress/integration/pages/LoginPage.js**

class LoginPage {

visit() {

cy.visit(Cypress.env("baseUrl"));

}

enterUsername(username) {

cy.get("#username").type(username);

}

enterPassword(password) {

cy.get("#password").type(password);

}

clickLogin() {

cy.get("#login-button").click();

}

}

export default LoginPage;

**4. Test Cases**

Use the test data and keywords in test cases.

**cypress/integration/test\_cases/login.spec.js**

import LoginPage from "../pages/LoginPage";

describe("Login Tests", () => {

const loginPage = new LoginPage();

beforeEach(() => {

loginPage.visit();

});

it("Data-Driven Login Tests", () => {

cy.fixture("test\_data").then((data) => {

data.loginData.forEach((testData) => {

loginPage.enterUsername(testData.username);

loginPage.enterPassword(testData.password);

loginPage.clickLogin();

if (testData.expected === "success") {

cy.verifyTitle("Dashboard");

} else {

cy.get(".error-message").should("contain", "Invalid credentials");

}

});

});

});

});

**5. Configuration Management**

Centralize configurations.

**cypress.config.js**

const { defineConfig } = require("cypress");

module.exports = defineConfig({

env: {

baseUrl: "https://example.com",

},

reporter: "mochawesome",

reporterOptions: {

reportDir: "reports",

overwrite: false,

html: true,

json: true,

},

});

**cypress.env.json**

{

"baseUrl": "https://example.com"

}

**6. Reporting**

Install **Mochawesome** for reporting:

npm install mochawesome --save-dev

Run tests with reporting:

npx cypress run --reporter mochawesome

**Advantages of a Hybrid Framework in Cypress**

1. **Modularity**:
   * Reusable components (custom commands, page objects, and test data).
2. **Scalability**:
   * Easily extendable for large projects or new test cases.
3. **Flexibility**:
   * Combines multiple testing approaches tailored to the project’s needs.
4. **Ease of Maintenance**:
   * Centralized configurations and reusable code reduce overhead.
5. **Rich Reporting**:
   * Detailed reports with screenshots and logs.

**Best Practices**

1. **Error Handling**:
   * Add robust assertions and validations in reusable keywords.
2. **Version Control**:
   * Use Git for collaborative development and version tracking.
3. **CI/CD Integration**:
   * Automate Cypress tests using Jenkins, GitHub Actions, or CircleCI.
4. **Documentation**:
   * Maintain clear documentation for framework setup and usage.

Would you like assistance in setting up a specific part of this Cypress hybrid framework or integrating additional features, such as API testing or CI/CD?