Professional Test Automation Framework - Complete Guide

Table of Contents

Part 1: Project Setup and Structure

Required Dependencies

Create a `requirements.txt` file with the following dependencies:

```
# Core testing dependencies
selenium==4.15.2
pytest==7.4.3
pytest-xdist==3.3.1
pytest-html==4.1.1

# Reporting and logging
allure-pytest==2.13.2
pytest-rerunfailures==12.0

# Data handling
openpyxl==3.1.2
pandas==2.1.3

# Configuration and utilities
pyyaml==6.0.1
python-dotenv==1.0.0
```

```
# WebDriver management
webdriver-manager==4.0.1

# Additional utilities
colorlog==6.7.0
```

Project Directory Structure

```
test_automation_framework/
│
├── config/
│   ├── __init__.py
│   ├── config.yaml
│   └── environment.py
│
├── pages/
│   ├── __init__.py
│   ├── base_page.py
│   └── login_page.py
│
├── tests/
│   ├── __init__.py
│   ├── base_test.py
│   └── test_login.py
│
├── utils/
│   ├── __init__.py
│   ├── logger.py
│   ├── excel_reader.py
│   └── driver_manager.py
│
├── test_data/
│   ├── login_data.xlsx
│   └── test_urls.yaml
│
├── reports/
│   ├── allure-results/
│   └── logs/
│
├── requirements.txt
├── pytest.ini
├── Jenkinsfile
└── README.md
```

Directory Purpose:

- **config/**: Configuration files for different environments and settings
- **pages/**: Page Object Model classes representing web pages
- **tests/**: Test files and base test classes
- **utils/**: Utility functions for logging, data reading, driver management
- **test_data/**: Excel files and YAML files containing test data
- **reports/**: Generated reports and logs

Configuration Files

config/config.yaml

```
environments:
  dev:
    base_url: "https://dev.example.com"
    username: "dev_user"
    password: "dev_pass"

  staging:
    base_url: "https://staging.example.com"
    username: "staging_user"
    password: "staging_pass"

  prod:
    base_url: "https://example.com"
    username: "prod_user"
    password: "prod_pass"

browser:
  default: "chrome"
  headless: false
  implicit_wait: 10
  explicit_wait: 20
  page_load_timeout: 30

logging:
  level: "INFO"
  format: "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
  file_path: "reports/logs/test_execution.log"
```

pytest.ini

```
[tool:pytest]
```

```
markers =
    smoke: marks tests as smoke tests
    regression: marks tests as regression tests
    critical: marks tests as critical tests
    login: marks tests related to login functionality

addopts =
    --strict-markers
    --alluredir=reports/allure-results
    --html=reports/html_report.html
    --self-contained-html
    -v

testpaths = tests
python_files = test_*.py
python_classes = Test*
python_functions = test_*

log_cli = true
log_cli_level = INFO
log_cli_format = %(asctime)s [%(levelname)8s] %(name)s: %(message)s
log_cli_date_format = %Y-%m-%d %H:%M:%S
```

Part 2: Core Framework Components

BasePage Class

pages/base_page.py

```
"""
Base Page class containing common methods for all page objects.
This class implements the foundation of the Page Object Model pattern.
"""

import logging
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
from selenium.webdriver.common.action_chains import ActionChains
from selenium.common.exceptions import TimeoutException, NoSuchElementException
import allure
```

```
class BasePage:
    """
    Base class for all page objects. Contains common methods that can be used
    across all page objects to maintain DRY principle.
    """

    def __init__(self, driver):
        """
        Initialize BasePage with WebDriver instance.

        Args:
            driver: WebDriver instance
        """
        self.driver = driver
        self.wait = WebDriverWait(driver, 20)
        self.logger = logging.getLogger(__name__)

    @allure.step("Click element: {locator}")
    def click(self, locator, timeout=20):
        """
        Click on an element after waiting for it to be clickable.

        Args:
            locator: Tuple containing (By, value) for element location
            timeout: Maximum time to wait for element

        Returns:
            bool: True if click successful, False otherwise
        """
        try:
            element = WebDriverWait(self.driver, timeout).until(
                EC.element_to_be_clickable(locator)
            )
            element.click()
            self.logger.info(f"Successfully clicked element: {locator}")
            return True
        except TimeoutException:
            self.logger.error(f"Timeout waiting for clickable element: {locator}")
            return False
        except Exception as e:
            self.logger.error(f"Error clicking element {locator}: {str(e)}")
            return False
```

```
@allure.step("Enter text '{text}' into element: {locator}")
def send_keys(self, locator, text, timeout=20, clear_first=True):
    """
    Send keys to an element after waiting for it to be present.

    Args:
        locator: Tuple containing (By, value) for element location
        text: Text to enter
        timeout: Maximum time to wait for element
        clear_first: Whether to clear the field before entering text

    Returns:
        bool: True if text entry successful, False otherwise
    """
    try:
        element = WebDriverWait(self.driver, timeout).until(
            EC.presence_of_element_located(locator)
        )
        if clear_first:
            element.clear()
        element.send_keys(text)
        self.logger.info(f"Successfully entered text '{text}' into element: {locator}")
        return True
    except TimeoutException:
        self.logger.error(f"Timeout waiting for element: {locator}")
        return False
    except Exception as e:
        self.logger.error(f"Error entering text into element {locator}: {str(e)}")
        return False

@allure.step("Check if element is visible: {locator}")
def is_visible(self, locator, timeout=10):
    """
    Check if an element is visible on the page.

    Args:
        locator: Tuple containing (By, value) for element location
        timeout: Maximum time to wait for element

    Returns:
        bool: True if element is visible, False otherwise
    """
    try:
        WebDriverWait(self.driver, timeout).until(
```

```
            EC.visibility_of_element_located(locator)
        )
        self.logger.info(f"Element is visible: {locator}")
        return True
    except TimeoutException:
        self.logger.info(f"Element is not visible: {locator}")
        return False

@allure.step("Get text from element: {locator}")
def get_text(self, locator, timeout=20):
    """
    Get text from an element.

    Args:
        locator: Tuple containing (By, value) for element location
        timeout: Maximum time to wait for element

    Returns:
        str: Text content of the element, empty string if not found
    """
    try:
        element = WebDriverWait(self.driver, timeout).until(
            EC.presence_of_element_located(locator)
        )
        text = element.text
        self.logger.info(f"Retrieved text '{text}' from element: {locator}")
        return text
    except TimeoutException:
        self.logger.error(f"Timeout waiting for element: {locator}")
        return ""
    except Exception as e:
        self.logger.error(f"Error getting text from element {locator}: {str(e)}")
        return ""

@allure.step("Get page title")
def get_title(self):
    """
    Get the title of the current page.

    Returns:
        str: Page title
    """
    title = self.driver.title
    self.logger.info(f"Page title: {title}")
```

```
    return title

@allure.step("Navigate to URL: {url}")
def navigate_to(self, url):
    """
    Navigate to a specific URL.

    Args:
        url: URL to navigate to
    """
    try:
        self.driver.get(url)
        self.logger.info(f"Successfully navigated to: {url}")
    except Exception as e:
        self.logger.error(f"Error navigating to {url}: {str(e)}")

@allure.step("Wait for element to be present: {locator}")
def wait_for_element(self, locator, timeout=20):
    """
    Wait for an element to be present in the DOM.

    Args:
        locator: Tuple containing (By, value) for element location
        timeout: Maximum time to wait for element

    Returns:
        WebElement or None: The element if found, None otherwise
    """
    try:
        element = WebDriverWait(self.driver, timeout).until(
            EC.presence_of_element_located(locator)
        )
        self.logger.info(f"Element found: {locator}")
        return element
    except TimeoutException:
        self.logger.error(f"Timeout waiting for element: {locator}")
        return None

@allure.step("Scroll to element: {locator}")
def scroll_to_element(self, locator):
    """
    Scroll to an element on the page.

    Args:
```

```
        locator: Tuple containing (By, value) for element location
    """
    try:
        element = self.driver.find_element(*locator)
        self.driver.execute_script("arguments[0].scrollIntoView(true);", element)
        self.logger.info(f"Scrolled to element: {locator}")
    except Exception as e:
        self.logger.error(f"Error scrolling to element {locator}: {str(e)}")

@allure.step("Take screenshot")
def take_screenshot(self, filename=None):
    """
    Take a screenshot of the current page.

    Args:
        filename: Optional filename for the screenshot

    Returns:
        str: Path to the saved screenshot
    """
    if not filename:
        from datetime import datetime
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = f"screenshot_{timestamp}.png"

    screenshot_path = f"reports/screenshots/{filename}"
    try:
        self.driver.save_screenshot(screenshot_path)
        self.logger.info(f"Screenshot saved: {screenshot_path}")
        return screenshot_path
    except Exception as e:
        self.logger.error(f"Error taking screenshot: {str(e)}")
        return None
```

BaseTest Class

tests/base_test.py

```
"""
Base Test class containing common setup and teardown methods.
This class implements the foundation for all test classes.
"""
```

```
import pytest
import logging
from selenium import webdriver
from selenium.webdriver.chrome.options import Options as ChromeOptions
from selenium.webdriver.firefox.options import Options as FirefoxOptions
from webdriver_manager.chrome import ChromeDriverManager
from webdriver_manager.firefox import GeckoDriverManager
from selenium.webdriver.chrome.service import Service as ChromeService
from selenium.webdriver.firefox.service import Service as FirefoxService
import allure
from utils.logger import setup_logger
from config.environment import Environment


class BaseTest:
    """
    Base class for all test classes. Contains common setup and teardown methods.
    """

    @pytest.fixture(scope="function", autouse=True)
    def setup_and_teardown(self, request):
        """
        Setup and teardown fixture that runs before and after each test.

        Args:
            request: Pytest request object containing test information
        """
        # Setup logger
        self.logger = setup_logger()

        # Get environment configuration
        self.env = Environment()

        # Setup WebDriver
        self.driver = self._setup_driver()

        # Make driver available to test methods
        request.cls.driver = self.driver

        # Log test start
        test_name = request.node.name
        self.logger.info(f"Starting test: {test_name}")

        with allure.step(f"Test Setup: {test_name}"):
```

```
            self.driver.maximize_window()
            self.driver.implicitly_wait(self.env.config['browser']['implicit_wait'])
            self.driver.set_page_load_timeout(self.env.config['browser']['page_load_timeout'])

        yield  # This is where the test runs

        # Teardown
        with allure.step("Test Teardown"):
            self._capture_failure_evidence(request)
            self.logger.info(f"Finished test: {test_name}")
            if self.driver:
                self.driver.quit()

    def _setup_driver(self):
        """
        Setup WebDriver based on configuration.

        Returns:
            WebDriver: Configured WebDriver instance
        """
        browser = self.env.config['browser']['default'].lower()
        headless = self.env.config['browser']['headless']

        if browser == 'chrome':
            return self._setup_chrome_driver(headless)
        elif browser == 'firefox':
            return self._setup_firefox_driver(headless)
        else:
            raise ValueError(f"Unsupported browser: {browser}")

    def _setup_chrome_driver(self, headless=False):
        """
        Setup Chrome WebDriver with options.

        Args:
            headless: Whether to run in headless mode

        Returns:
            WebDriver: Chrome WebDriver instance
        """
        options = ChromeOptions()

        if headless:
            options.add_argument('--headless')
```

```
        options.add_argument('--no-sandbox')
        options.add_argument('--disable-dev-shm-usage')
        options.add_argument('--disable-gpu')
        options.add_argument('--window-size=1920,1080')
        options.add_argument('--disable-extensions')
        options.add_argument('--disable-web-security')
        options.add_argument('--allow-running-insecure-content')

        service = ChromeService(ChromeDriverManager().install())
        driver = webdriver.Chrome(service=service, options=options)

        self.logger.info("Chrome WebDriver initialized successfully")
        return driver

    def _setup_firefox_driver(self, headless=False):
        """
        Setup Firefox WebDriver with options.

        Args:
            headless: Whether to run in headless mode

        Returns:
            WebDriver: Firefox WebDriver instance
        """
        options = FirefoxOptions()

        if headless:
            options.add_argument('--headless')

        options.add_argument('--width=1920')
        options.add_argument('--height=1080')

        service = FirefoxService(GeckoDriverManager().install())
        driver = webdriver.Firefox(service=service, options=options)

        self.logger.info("Firefox WebDriver initialized successfully")
        return driver

    def _capture_failure_evidence(self, request):
        """
        Capture screenshot and page source on test failure.

        Args:
```

```
        request: Pytest request object
    """
    if request.node.rep_call.failed:
        test_name = request.node.name

        # Take screenshot
        screenshot_path = f"reports/screenshots/failed_{test_name}.png"
        try:
            self.driver.save_screenshot(screenshot_path)
            allure.attach.file(screenshot_path, name="Failure Screenshot",
                               attachment_type=allure.attachment_type.PNG)
            self.logger.error(f"Test failed. Screenshot saved: {screenshot_path}")
        except Exception as e:
            self.logger.error(f"Failed to capture screenshot: {str(e)}")

        # Capture page source
        try:
            page_source = self.driver.page_source
            allure.attach(page_source, name="Page Source",
                          attachment_type=allure.attachment_type.HTML)
        except Exception as e:
            self.logger.error(f"Failed to capture page source: {str(e)}")

@pytest.hookimpl(hookwrapper=True, tryfirst=True)
def pytest_runtest_makereport(self, item, call):
    """
    Pytest hook to capture test results for failure handling.
    """
    outcome = yield
    rep = outcome.get_result()
    setattr(item, "rep_" + rep.when, rep)
    return rep
```

Part 3: Implementing Page Object Model

Sample Login Page

pages/login_page.py

```
"""
Login Page Object Model implementation.
Contains all elements and methods related to the login functionality.
```

```
"""

from selenium.webdriver.common.by import By
import allure
from pages.base_page import BasePage


class LoginPage(BasePage):
    """
    Login Page class implementing Page Object Model pattern.
    Contains locators and methods for login page interactions.
    """

    # Page locators
    USERNAME_FIELD = (By.ID, "username")
    PASSWORD_FIELD = (By.ID, "password")
    LOGIN_BUTTON = (By.ID, "login-button")
    ERROR_MESSAGE = (By.CLASS_NAME, "error-message")
    FORGOT_PASSWORD_LINK = (By.LINK_TEXT, "Forgot Password?")
    REMEMBER_ME_CHECKBOX = (By.ID, "remember-me")
    LOGIN_FORM = (By.ID, "login-form")
    LOGO = (By.CLASS_NAME, "logo")

    # Success indicators
    DASHBOARD_HEADER = (By.CLASS_NAME, "dashboard-header")
    USER_PROFILE_MENU = (By.ID, "user-profile")

    def __init__(self, driver):
        """
        Initialize LoginPage with WebDriver instance.

        Args:
            driver: WebDriver instance
        """
        super().__init__(driver)
        self.page_url = "/login"

    @allure.step("Navigate to login page")
    def navigate_to_login_page(self, base_url):
        """
        Navigate to the login page.

        Args:
            base_url: Base URL of the application
```

```
    """
    full_url = base_url + self.page_url
    self.navigate_to(full_url)
    self.wait_for_page_to_load()

@allure.step("Wait for login page to load")
def wait_for_page_to_load(self):
    """
    Wait for the login page to fully load.

    Returns:
        bool: True if page loaded successfully, False otherwise
    """
    return self.is_visible(self.LOGIN_FORM, timeout=10)

@allure.step("Enter username: {username}")
def enter_username(self, username):
    """
    Enter username in the username field.

    Args:
        username: Username to enter

    Returns:
        bool: True if successful, False otherwise
    """
    return self.send_keys(self.USERNAME_FIELD, username)

@allure.step("Enter password")
def enter_password(self, password):
    """
    Enter password in the password field.

    Args:
        password: Password to enter

    Returns:
        bool: True if successful, False otherwise
    """
    return self.send_keys(self.PASSWORD_FIELD, password)

@allure.step("Click login button")
def click_login_button(self):
    """
```

```
    Click the login button.

    Returns:
        bool: True if successful, False otherwise
    """
    return self.click(self.LOGIN_BUTTON)

@allure.step("Check remember me checkbox")
def check_remember_me(self):
    """
    Check the remember me checkbox.

    Returns:
        bool: True if successful, False otherwise
    """
    return self.click(self.REMEMBER_ME_CHECKBOX)

@allure.step("Click forgot password link")
def click_forgot_password(self):
    """
    Click the forgot password link.

    Returns:
        bool: True if successful, False otherwise
    """
    return self.click(self.FORGOT_PASSWORD_LINK)

@allure.step("Get error message")
def get_error_message(self):
    """
    Get the error message displayed on the page.

    Returns:
        str: Error message text
    """
    return self.get_text(self.ERROR_MESSAGE)

@allure.step("Check if error message is displayed")
def is_error_message_displayed(self):
    """
    Check if error message is displayed.

    Returns:
        bool: True if error message is visible, False otherwise
```

```
        """
        return self.is_visible(self.ERROR_MESSAGE, timeout=5)

    @allure.step("Check if login was successful")
    def is_login_successful(self):
        """
        Check if login was successful by looking for dashboard elements.

        Returns:
            bool: True if login successful, False otherwise
        """
        return (self.is_visible(self.DASHBOARD_HEADER, timeout=10) or
                self.is_visible(self.USER_PROFILE_MENU, timeout=10))

    @allure.step("Perform login with credentials")
    def login(self, username, password, remember_me=False):
        """
        Perform complete login process.

        Args:
            username: Username to use for login
            password: Password to use for login
            remember_me: Whether to check remember me checkbox

        Returns:
            bool: True if login process completed successfully, False otherwise
        """
        try:
            if not self.enter_username(username):
                return False

            if not self.enter_password(password):
                return False

            if remember_me:
                self.check_remember_me()

            if not self.click_login_button():
                return False

            self.logger.info(f"Login attempt completed for user: {username}")
            return True

        except Exception as e:
```

```
            self.logger.error(f"Error during login process: {str(e)}")
            return False

    @allure.step("Verify login page elements")
    def verify_page_elements(self):
        """
        Verify that all expected elements are present on the login page.

        Returns:
            dict: Dictionary containing verification results for each element
        """
        elements_to_verify = {
            'username_field': self.USERNAME_FIELD,
            'password_field': self.PASSWORD_FIELD,
            'login_button': self.LOGIN_BUTTON,
            'forgot_password_link': self.FORGOT_PASSWORD_LINK,
            'remember_me_checkbox': self.REMEMBER_ME_CHECKBOX,
            'logo': self.LOGO
        }

        verification_results = {}
        for element_name, locator in elements_to_verify.items():
            verification_results[element_name] = self.is_visible(locator, timeout=5)

        self.logger.info(f"Page elements verification: {verification_results}")
        return verification_results
```

Part 4: Writing Tests with Pytest and TDD

Sample Test File

tests/test_login.py

```
"""
Login functionality test cases.
Demonstrates TDD approach and proper test structure.
"""

import pytest
import allure
from tests.base_test import BaseTest
from pages.login_page import LoginPage
```

```
from utils.excel_reader import ExcelReader
from config.environment import Environment


@allure.feature("Authentication")
@allure.story("User Login")
class TestLogin(BaseTest):
    """
    Test class for login functionality.
    Following TDD approach: Write test first, then implement page objects.
    """

    def setup_method(self):
        """
        Setup method that runs before each test method.
        """
        self.env = Environment()
        self.login_page = LoginPage(self.driver)
        self.base_url = self.env.get_base_url()

    @allure.title("Test successful login with valid credentials")
    @allure.severity(allure.severity_level.CRITICAL)
    @pytest.mark.smoke
    @pytest.mark.critical
    def test_successful_login(self):
        """
        Test Case: Verify successful login with valid credentials

        Steps:
        1. Navigate to login page
        2. Enter valid username and password
        3. Click login button
        4. Verify successful login
        """
        with allure.step("Navigate to login page"):
            self.login_page.navigate_to_login_page(self.base_url)
            assert self.login_page.wait_for_page_to_load(), "Login page did not load properly"

        with allure.step("Verify all page elements are present"):
            page_elements = self.login_page.verify_page_elements()
            assert all(page_elements.values()), f"Missing page elements: {page_elements}"

        with allure.step("Perform login with valid credentials"):
            username = self.env.get_username()
```

```
            password = self.env.get_password()

            login_successful = self.login_page.login(username, password)
            assert login_successful, "Login process failed"

        with allure.step("Verify successful login"):
            assert self.login_page.is_login_successful(), "Login was not successful"

        with allure.step("Verify page title after login"):
            expected_title = "Dashboard"  # Adjust based on your application
            actual_title = self.login_page.get_title()
            assert expected_title in actual_title, f"Expected '{expected_title}' in title, but got
'{actual_title}'"

    @allure.title("Test login failure with invalid credentials")
    @allure.severity(allure.severity_level.HIGH)
    @pytest.mark.smoke
    def test_invalid_login(self):
        """
        Test Case: Verify login failure with invalid credentials

        Steps:
        1. Navigate to login page
        2. Enter invalid username and password
        3. Click login button
        4. Verify error message is displayed
        """
        with allure.step("Navigate to login page"):
            self.login_page.navigate_to_login_page(self.base_url)
            assert self.login_page.wait_for_page_to_load(), "Login page did not load properly"

        with allure.step("Attempt login with invalid credentials"):
            invalid_username = "invalid_user"
            invalid_password = "invalid_password"

            login_attempted = self.login_page.login(invalid_username, invalid_password)
            assert login_attempted, "Login attempt failed"

        with allure.step("Verify error message is displayed"):
            assert self.login_page.is_error_message_displayed(), "Error message is not displayed"

            error_message = self.login_page.get_error_message()
            expected_error_messages = ["Invalid credentials", "Login failed", "Incorrect username or
password"]
```

```
            assert any(expected_msg.lower() in error_message.lower() for expected_msg in
expected_error_messages), \
                f"Expected error message not found. Actual message: '{error_message}'"

        with allure.step("Verify user is not logged in"):
            assert not self.login_page.is_login_successful(), "User should not be logged in with
invalid credentials"

    @allure.title("Test empty username field validation")
    @allure.severity(allure.severity_level.MEDIUM)
    @pytest.mark.regression
    def test_empty_username(self):
        """
        Test Case: Verify validation when username field is empty
        """
        with allure.step("Navigate to login page"):
            self.login_page.navigate_to_login_page(self.base_url)
            assert self.login_page.wait_for_page_to_load(), "Login page did not load properly"

        with allure.step("Enter only password and attempt login"):
            password = self.env.get_password()

            self.login_page.enter_password(password)
            self.login_page.click_login_button()

        with allure.step("Verify appropriate error message"):
            assert self.login_page.is_error_message_displayed(), "Error message should be
displayed for empty username"

            error_message = self.login_page.get_error_message()
            expected_keywords = ["username", "required", "empty"]

            assert any(keyword.lower() in error_message.lower() for keyword in
expected_keywords), \
                f"Error message should mention username requirement. Actual: '{error_message}'"

    @allure.title("Test empty password field validation")
    @allure.severity(allure.severity_level.MEDIUM)
    @pytest.mark.regression
    def test_empty_password(self):
        """
        Test Case: Verify validation when password field is empty
        """
```

```
        with allure.step("Navigate to login page"):
            self.login_page.navigate_to_login_page(self.base_url)
            assert self.login_page.wait_for_page_to_load(), "Login page did not load properly"

        with allure.step("Enter only username and attempt login"):
            username = self.env.get_username()

            self.login_page.enter_username(username)
            self.login_page.click_login_button()

        with allure.step("Verify appropriate error message"):
            assert self.login_page.is_error_message_displayed(), "Error message should be
displayed for empty password"

            error_message = self.login_page.get_error_message()
            expected_keywords = ["password", "required", "empty"]

            assert any(keyword.lower() in error_message.lower() for keyword in
expected_keywords),
```