kasik1: I am using PyTest, Django, API, Selenium. Please find the attachment above test zip file our framework folder structure. High level we are using the confest file for every test cases

Workspace: Collecting workspace informationFiltering to most relevant informationYour framework structure and the use of conftest.py files for managing fixtures across test cases is a common and effective approach in PyTest. Based on the provided conftest.py file, here are some observations and suggestions:

### Observations:

1. \*\*Global Driver Fixture\*\*:

- The `global\_driver` fixture initializes a Selenium WebDriver instance, navigates to the `setup\_server\_url`, and logs in using credentials fetched from `get\_project\_credentials`.

- It ensures proper cleanup by closing and quitting the driver after the session.

2. \*\*User Base Driver Fixture\*\*:

- The `user\_base\_driver` fixture is similar to `global\_driver` but does not perform login operations. It is likely used for tests that require a driver but do not need an authenticated session.

3. \*\*Plugins\*\*:

- The `pytest\_plugins` list includes custom plugins for API testing, global fixtures, reporting, and hooks for test reporting. This modular approach helps in maintaining a clean and reusable test framework.

4. \*\*Django Integration\*\*:

- The use of `django\_api\_config` suggests that the framework integrates with Django for fetching project-specific configurations, such as credentials.

5. \*\*Selenium Integration\*\*:

- Selenium is used for UI testing, with WebDriver instances being created dynamically based on the `test\_mode`, `browser`, and `ui\_mode` parameters.

### Suggestions:

1. \*\*Parameterization\*\*:

- If you are testing across multiple browsers or environments, consider using `pytest.mark.parametrize` or `pytest\_addoption` to pass browser and environment configurations dynamically.

2. \*\*Fixture Scope\*\*:

- The `scope='session'` for the fixtures is appropriate for tests that require a single driver instance per test session. If some tests require a fresh driver instance, consider adding `scope='function'` fixtures.

3. \*\*Error Handling\*\*:

- Add error handling in the `global\_driver` and `user\_base\_driver` fixtures to ensure the driver is closed properly even if an exception occurs during test execution.

4. \*\*Reusable Components\*\*:

- If you have repetitive actions (e.g., login, navigation), encapsulate them in helper functions or Page Object Model (POM) classes to improve maintainability.

5. \*\*Test Data Management\*\*:

- Use `pytest.fixture` to load test data dynamically from external files (e.g., JSON, CSV) for better test coverage and flexibility.

6. \*\*Django Test Integration\*\*:

- If you are using Django's test database, ensure that your tests are wrapped in Django's test transaction to maintain database isolation.

7. \*\*Reporting\*\*:

- Leverage plugins like `pytest-html` or `pytest-allure` for generating detailed test reports, especially for Selenium-based UI tests.

8. \*\*Parallel Execution\*\*:

- If you have a large number of tests, consider using `pytest-xdist` for parallel execution to reduce test runtime.

Let me know if you need help implementing any of these suggestions or have specific questions about your framework!

kasik1: yes Provide recommendations for architecture improvement and improving test speeds/run time

GitHub Copilot: Here are recommendations to improve the architecture of your test framework and reduce test execution time:

---

### \*\*1. Optimize Fixture Scope\*\*

- \*\*Current Issue\*\*: Both `global\_driver` and `user\_base\_driver` have a `scope='session'`, meaning the same driver instance is shared across all tests in the session. While this is efficient for some tests, it can cause bottlenecks if tests are dependent on state or require isolation.

- \*\*Improvement\*\*:

- Use `scope='function'` for tests that require a fresh driver instance.

- Consider splitting tests into separate test sessions if they require different driver configurations.

```python

@pytest.fixture(scope='function') # Change scope to function for isolated tests

def isolated\_driver(setup\_server\_url, browser, test\_mode, ui\_mode):

driver = Driver.create\_driver(test\_mode, browser, ui\_mode=ui\_mode)

driver.get(setup\_server\_url)

yield driver

driver.close()

driver.quit()

```

---

### \*\*2. Parallel Test Execution\*\*

- \*\*Current Issue\*\*: Tests are likely running sequentially, which increases runtime, especially for Selenium-based UI tests.

- \*\*Improvement\*\*: Use `pytest-xdist` to run tests in parallel. This is particularly useful for Selenium tests, as each test can run in its own browser instance.

\*\*Steps\*\*:

1. Install `pytest-xdist`:

```bash

pip install pytest-xdist

```

2. Run tests with parallel workers:

```bash

pytest -n auto

```

The `-n auto` flag automatically determines the number of workers based on available CPU cores.

---

### \*\*3. Use Headless Browsers\*\*

- \*\*Current Issue\*\*: Running tests with a full browser UI can slow down execution.

- \*\*Improvement\*\*: Use headless mode for Selenium WebDriver to speed up tests, especially for non-visual validations.

```python

# Modify Driver.create\_driver to include headless mode

options = webdriver.ChromeOptions()

options.add\_argument('--headless')

options.add\_argument('--disable-gpu')

driver = webdriver.Chrome(options=options)

```

---

### \*\*4. Reduce Redundant Logins\*\*

- \*\*Current Issue\*\*: The `global\_driver` fixture logs in for every session, which can be time-consuming.

- \*\*Improvement\*\*: Cache the login session or use API-based authentication to bypass the UI login process.

```python

@pytest.fixture(scope='session')

def authenticated\_driver(setup\_server\_url, browser, test\_mode, ui\_mode):

driver = Driver.create\_driver(test\_mode, browser, ui\_mode=ui\_mode)

driver.get(setup\_server\_url)

# Use API to authenticate and set cookies

session\_cookies = get\_authentication\_cookies()

for cookie in session\_cookies:

driver.add\_cookie(cookie)

yield driver

driver.close()

driver.quit()

```

---

### \*\*5. Modularize Page Interactions\*\*

- \*\*Current Issue\*\*: The `Login` class is directly used in the fixture, which can lead to tight coupling.

- \*\*Improvement\*\*: Implement the Page Object Model (POM) to encapsulate page interactions. This makes tests more maintainable and reusable.

```python

# Example: Login Page Object

class LoginPage:

def \_\_init\_\_(self, driver):

self.driver = driver

def login(self, username, password):

self.driver.find\_element(By.ID, "username").send\_keys(username)

self.driver.find\_element(By.ID, "password").send\_keys(password)

self.driver.find\_element(By.ID, "login-button").click()

```

---

### \*\*6. Use Smart Test Selection\*\*

- \*\*Current Issue\*\*: Running all tests every time can be inefficient.

- \*\*Improvement\*\*: Use `pytest` markers and `pytest-cache` to run only relevant tests.

- Run only failed tests:

```bash

pytest --lf

```

- Run tests based on markers:

```bash

pytest -m "smoke"

```

---

### \*\*7. Optimize Database Usage\*\*

- \*\*Current Issue\*\*: If tests interact with the database, setup and teardown operations can be slow.

- \*\*Improvement\*\*:

- Use Django's `TransactionTestCase` or `pytest-django` to leverage in-memory databases for faster test execution.

- Mock database calls for tests that do not require actual database interactions.

---

### \*\*8. Improve Reporting\*\*

- \*\*Current Issue\*\*: Lack of detailed reporting can slow down debugging.

- \*\*Improvement\*\*: Use plugins like `pytest-html` or `allure-pytest` for detailed HTML reports.

\*\*Install and Use `pytest-html`\*\*:

```bash

pip install pytest-html

pytest --html=report.html

```

---

### \*\*9. Use Browser Containers\*\*

- \*\*Current Issue\*\*: Managing browser dependencies locally can be cumbersome.

- \*\*Improvement\*\*: Use Docker containers with Selenium Grid for consistent and scalable browser testing.

\*\*Example\*\*:

1. Set up Selenium Grid using Docker:

```bash

docker run -d -p 4444:4444 --name selenium-hub selenium/hub

docker run -d --link selenium-hub:hub selenium/node-chrome

```

2. Update `Driver.create\_driver` to connect to the Selenium Grid.

---

### \*\*10. Profile and Optimize Test Code\*\*

- \*\*Current Issue\*\*: Inefficient test code can increase runtime.

- \*\*Improvement\*\*: Use Python's `cProfile` or `pytest-profiling` to identify bottlenecks in your tests.

\*\*Install and Use `pytest-profiling`\*\*:

```bash

pip install pytest-profiling

pytest --profile

```

---

By implementing these recommendations, you can significantly improve the architecture, maintainability, and execution speed of your test framework. Let me know if you'd like help with any specific implementation!

kasik1: can crate word document with all details and share here link for download

GitHub Copilot: Sorry, I can't assist with that.

kasik1: we are using parall only

GitHub Copilot: That's great to hear that you're already using parallel execution! Here are additional recommendations to further optimize your test framework and improve test speeds:

---

### \*\*1. Optimize Parallel Execution\*\*

- \*\*Issue\*\*: Even with parallel execution, resource contention (e.g., database, browser instances) can slow down tests.

- \*\*Improvement\*\*:

- \*\*Limit Resource Contention\*\*: Ensure that tests running in parallel do not share state or resources like databases or files.

- \*\*Distribute Tests Evenly\*\*: Use `pytest-xdist`'s `--dist` option to distribute tests more efficiently:

```bash

pytest -n auto --dist loadscope

```

The `loadscope` strategy ensures that tests in the same module are run on the same worker, reducing setup overhead.

---

### \*\*2. Use Lightweight Fixtures\*\*

- \*\*Issue\*\*: Session-scoped fixtures like `global\_driver` and `user\_base\_driver` can become bottlenecks if they perform heavy operations.

- \*\*Improvement\*\*:

- Use function-scoped fixtures for tests that don't need a shared driver.

- Cache expensive operations (e.g., login) to avoid repeating them unnecessarily.

---

### \*\*3. Reduce Selenium Overhead\*\*

- \*\*Issue\*\*: Selenium tests are inherently slower due to browser interactions.

- \*\*Improvement\*\*:

- Use \*\*headless browsers\*\* to reduce UI rendering overhead.

- Use \*\*explicit waits\*\* instead of implicit waits to minimize delays.

- For tests that don't require full browser functionality, consider using \*\*API calls\*\* or tools like \*\*Playwright\*\*, which are faster than Selenium.

---

### \*\*4. Optimize Database Usage\*\*

- \*\*Issue\*\*: Database setup and teardown can slow down tests.

- \*\*Improvement\*\*:

- Use \*\*transaction rollbacks\*\* instead of recreating the database for every test.

- Use \*\*in-memory databases\*\* (e.g., SQLite) for tests that don't require full database functionality.

---

### \*\*5. Smart Test Selection\*\*

- \*\*Issue\*\*: Running all tests every time can be inefficient.

- \*\*Improvement\*\*:

- Use `pytest`'s `--lf` (last failed) or `--ff` (failed first) options to prioritize failed tests:

```bash

pytest --lf

```

- Use `pytest` markers to run specific subsets of tests:

```bash

pytest -m "smoke"

```

---

### \*\*6. Improve Reporting\*\*

- \*\*Issue\*\*: Debugging failures can be time-consuming without detailed reports.

- \*\*Improvement\*\*:

- Use `pytest-html` or `allure-pytest` for detailed HTML reports.

- Include screenshots for Selenium test failures:

```python

@pytest.hookimpl(tryfirst=True, hookwrapper=True)

def pytest\_runtest\_makereport(item, call):

outcome = yield

report = outcome.get\_result()

if report.when == "call" and report.failed:

driver = item.funcargs.get('global\_driver')

if driver:

driver.save\_screenshot(f"{item.name}.png")

```

---

### \*\*7. Use Docker for Isolation\*\*

- \*\*Issue\*\*: Running tests locally can lead to inconsistent results due to environment differences.

- \*\*Improvement\*\*:

- Use Docker containers for consistent test environments.

- For Selenium, use \*\*Selenium Grid\*\* with Docker to manage browser instances efficiently.

---

### \*\*8. Profile and Optimize Tests\*\*

- \*\*Issue\*\*: Some tests may be slower due to inefficient code or unnecessary steps.

- \*\*Improvement\*\*:

- Use `pytest-profiling` to identify slow tests:

```bash

pip install pytest-profiling

pytest --profile

```

- Refactor slow tests to remove redundant steps.

---

By combining these strategies with your existing parallel execution setup, you can further reduce test runtime and improve efficiency. Let me know if you'd like help implementing any of these suggestions!

improvments

1. cenralized

import os

import pytest

from utils.csv\_reader import Csv # Replace with your actual CSV reader utility

@pytest.fixture(scope='session')

def spreadsheet():

"""Load the test data from a CSV file."""

path = os.path.dirname(os.path.abspath(\_\_file\_\_))

file\_path = os.path.join(path, 'test\_data.csv')

return Csv(file\_path)

@pytest.fixture(scope='session')

def setup\_server\_url(spreadsheet, testing\_environment):

"""Set up the server URL based on the testing environment."""

return testing\_environment(spreadsheet)

@pytest.fixture(scope='session')

def test\_data(spreadsheet):

"""Provide test data from the spreadsheet."""

return spreadsheet.get\_data()