PyTest + Django Automation Framework Overview

## ✅ PyTest + Django Test Automation Framework (Architecture Overview)  
  
### 📁 Folder Breakdown (from `tests/`):  
  
- `test\_cases/` – Actual UI/API test scripts  
 - ✅ Includes `async\_api\_test.py` (new POC)  
 - ✅ Includes `sample\_ui\_test.py` (headless + faker demo)  
- `pom/` – Page Object Model classes (Selenium, API)  
- `fixtures/` – Reusable setup and data injectors  
- `hooks/` – Pre/post-test actions (e.g., DB connect, cleanup)  
- `commons/` – Logging, data readers, shared functions  
 - ✅ Added `faker\_utils.py`, `json\_loader.py`  
- `data/` – ✅ New: Contains `test\_case\_metadata.json` (for test metadata)  
- `conftest.py` – ✅ Moved/Consolidated: Central root-level configuration and fixtures  
  
### ⚙️ Execution Flow:  
...

import aiohttp

import asyncio

import urllib3

urllib3.disable\_warnings(category=urllib3.exceptions.InsecureRequestWarning)

class BaseEndPoint:

def \_\_init\_\_(self, session, server\_url, endpoint=''):

"""

:param session: Fixture that will authenticate user's credentials

:param server\_url: Server on which the tests will execute

:param endpoint: API endpoint

"""

self.session = session

self.server = server\_url

self.endpoint = endpoint

self.headers = {}

self.body\_request = None

self.query\_parameters = None

self.path\_parameters = None

self.attachment\_file\_path = None

self.attachment\_file\_name = None

async def async\_get(self, \*args, \*\*kwargs):

"""

Asynchronous GET request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

kasik1: you can implment ayncy api here

GitHub Copilot: To implement asynchronous API methods in the `BaseEndPoint` class, you can use Python's `asyncio` and `aiohttp` libraries. Below is an example of how you can refactor the `BaseEndPoint` class to include asynchronous versions of the `get`, `post`, `put`, `patch`, and `delete` methods.

```python

import aiohttp

import asyncio

import urllib3

urllib3.disable\_warnings(category=urllib3.exceptions.InsecureRequestWarning)

class BaseEndPoint:

def \_\_init\_\_(self, session, server\_url, endpoint=''):

"""

:param session: Fixture that will authenticate user's credentials

:param server\_url: Server on which the tests will execute

:param endpoint: API endpoint

"""

self.session = session

self.server = server\_url

self.endpoint = endpoint

self.headers = {}

self.body\_request = None

self.query\_parameters = None

self.path\_parameters = None

self.attachment\_file\_path = None

self.attachment\_file\_name = None

async def async\_get(self, \*args, \*\*kwargs):

"""

Asynchronous GET request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

if kwargs:

request\_url = self.set\_query\_parameters(request\_url, kwargs)

async with aiohttp.ClientSession() as session:

async with session.get(request\_url, headers=self.headers, ssl=False) as response:

return await response.json()

async def async\_post(self, \*args, \*\*kwargs):

"""

Asynchronous POST request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

headers = {'Content-type': 'application/json'}

headers.update(self.headers)

async with aiohttp.ClientSession() as session:

async with session.post(request\_url, json=self.body\_request, headers=headers, ssl=False) as response:

return await response.json()

async def async\_put(self, \*args, \*\*kwargs):

"""

Asynchronous PUT request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

headers = {'Content-type': 'application/json'}

headers.update(self.headers)

async with aiohttp.ClientSession() as session:

async with session.put(request\_url, json=self.body\_request, headers=headers, ssl=False) as response:

return await response.json()

async def async\_patch(self, \*args, \*\*kwargs):

"""

Asynchronous PATCH request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

headers = {'Content-type': 'application/json'}

headers.update(self.headers)

async with aiohttp.ClientSession() as session:

async with session.patch(request\_url, json=self.body\_request, headers=headers, ssl=False) as response:

return await response.json()

async def async\_delete(self, \*args, \*\*kwargs):

"""

Asynchronous DELETE request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

async with aiohttp.ClientSession() as session:

async with session.delete(request\_url, headers=self.headers, ssl=False) as response:

return await response.json()

def format\_url(self, end\_point):

return self.server + end\_point

@staticmethod

def set\_query\_parameters(request\_url, query\_parameters):

request\_url += '?'

for query\_parameter in query\_parameters:

request\_url += '{}={}&'.format(query\_parameter, query\_parameters[query\_parameter])

return request\_url

```

### Key Changes:

1. \*\*Added `aiohttp` for asynchronous HTTP requests.\*\*

2. \*\*Implemented `async\_get`, `async\_post`, `async\_put`, `async\_patch`, and `async\_delete` methods.\*\*

3. \*\*Used `async with` for managing the `aiohttp.ClientSession`.\*\*

4. \*\*Ensured SSL verification is disabled (`ssl=False`) to match the behavior of `requests` with `verify=False`.\*\*

### Usage Example:

```python

import asyncio

async def main():

endpoint = BaseEndPoint(None, "https://example.com", "/api/resource")

response = await endpoint.async\_get()

print(response)

asyncio.run(main())

```

This approach allows you to perform non-blocking API calls, which is useful for high-performance applications.

kasik1: way callling it will change or smae

class Circuits(BaseEndPoint):

def \_\_init\_\_(self, session, server\_url, \*\*kwargs):

super(Circuits, self).\_\_init\_\_(session, server\_url, \*\*kwargs)

GitHub Copilot: The way you call the `Circuits` class will remain the same. The `Circuits` class inherits from `BaseEndPoint`, and its `\_\_init\_\_` method simply passes the arguments to the parent class using `super()`. Whether you use synchronous or asynchronous methods in `BaseEndPoint`, the instantiation of the `Circuits` class does not change.

### Example Usage (No Change):

```python

# Instantiate the Circuits class

circuits = Circuits(session, "https://example.com", endpoint="/api/circuits")

# Call synchronous methods

response = circuits.get()

print(response)

# Call asynchronous methods

import asyncio

async def main():

response = await circuits.async\_get()

print(response)

asyncio.run(main())

```

### Key Points:

- The `Circuits` class is a thin wrapper around `BaseEndPoint`, so its behavior depends entirely on the implementation of `BaseEndPoint`.

- If you add asynchronous methods to `BaseEndPoint`, you can directly use them with `Circuits` without modifying the `Circuits` class itself.

kasik1: just read fixures share draw backs still sync api calls its the best and insted of excel reusablites osn we can use

GitHub Copilot: ### Drawbacks of Current Fixtures with Synchronous API Calls

1. \*\*Blocking Nature of Synchronous Calls\*\*:

- The current implementation uses synchronous API calls (`requests` library). This can lead to performance bottlenecks, especially when dealing with multiple API calls or slow network responses. Each call blocks the execution until it completes, which is inefficient for high-throughput scenarios.

2. \*\*Limited Scalability\*\*:

- Synchronous calls are not ideal for scenarios where multiple test cases need to run concurrently. This can slow down test execution significantly, especially in CI/CD pipelines.

3. \*\*Tight Coupling with Excel\*\*:

- The `test\_data` fixture relies on an `excel` object to fetch test data. This approach is tightly coupled with Excel files, making it less reusable and harder to adapt to other data sources like databases, JSON, or CSV files.

4. \*\*Error Handling\*\*:

- The current implementation lacks robust error handling for API calls. For example, if an API call fails, there is no retry mechanism or detailed logging to help debug the issue.

5. \*\*Hardcoded Assumptions\*\*:

- The `test\_data` fixture assumes specific column names like `TEST\_TYPE` and `TEST\_CASE\_ID`. This makes the fixture less flexible and harder to adapt to different test data formats.

---

### Suggested Improvements

#### 1. \*\*Switch to Asynchronous API Calls\*\*

- Replace the synchronous `requests` library with `aiohttp` for non-blocking, asynchronous API calls. This will improve performance and scalability, especially for tests that involve multiple API calls.

Example:

```python

import aiohttp

import pytest

import asyncio

@pytest.fixture

async def async\_session():

async with aiohttp.ClientSession() as session:

yield session

```

Usage in tests:

```python

async def test\_api\_call(async\_session):

async with async\_session.get("https://example.com/api") as response:

assert response.status == 200

```

#### 2. \*\*Replace Excel with CSV or JSON for Reusability\*\*

- Instead of relying on Excel files, use CSV or JSON files for test data. These formats are easier to parse, more lightweight, and widely supported.

Example with CSV:

```python

import csv

import pytest

@pytest.fixture

def test\_data\_from\_csv():

with open("test\_data.csv", mode="r") as file:

reader = csv.DictReader(file)

data = [row for row in reader]

return data

```

Example with JSON:

```python

import json

import pytest

@pytest.fixture

def test\_data\_from\_json():

with open("test\_data.json", mode="r") as file:

data = json.load(file)

return data

```

#### 3. \*\*Add Retry Mechanism for API Calls\*\*

- Implement a retry mechanism for API calls to handle transient failures.

Example:

```python

import requests

from requests.adapters import HTTPAdapter

from urllib3.util.retry import Retry

def get\_session\_with\_retries():

session = requests.Session()

retries = Retry(total=3, backoff\_factor=1, status\_forcelist=[500, 502, 503, 504])

session.mount("http://", HTTPAdapter(max\_retries=retries))

session.mount("https://", HTTPAdapter(max\_retries=retries))

return session

```

#### 4. \*\*Parameterize Fixtures for Flexibility\*\*

- Use parameterized fixtures to make them more reusable across different test cases.

Example:

```python

@pytest.fixture

def test\_data(request):

data\_source = request.param

if data\_source == "csv":

# Load data from CSV

pass

elif data\_source == "json":

# Load data from JSON

pass

return data

```

#### 5. \*\*Improve Logging and Debugging\*\*

- Enhance logging to capture detailed information about API calls and test data processing.

Example:

```python

import logging

logger = logging.getLogger("test\_logger")

logger.setLevel(logging.DEBUG)

@pytest.fixture

def log\_api\_call():

def \_log(response):

logger.debug(f"URL: {response.url}")

logger.debug(f"Status Code: {response.status\_code}")

logger.debug(f"Response: {response.text}")

return \_log

```

---

### Why These Changes Are Better

1. \*\*Asynchronous API Calls\*\*:

- Non-blocking calls improve performance and allow concurrent execution of tests.

2. \*\*CSV/JSON for Test Data\*\*:

- These formats are lightweight, easier to parse, and more reusable across different test environments.

3. \*\*Retry Mechanism\*\*:

- Reduces test flakiness caused by transient network or server issues.

4. \*\*Parameterization\*\*:

- Makes fixtures more flexible and adaptable to different test scenarios.

5. \*\*Enhanced Logging\*\*:

- Provides better insights into test failures, making debugging easier.

By implementing these changes, the test suite will become more efficient, scalable, and maintainable.

Similar code found with 1 license type

if kwargs:

request\_url = self.set\_query\_parameters(request\_url, kwargs)

async with aiohttp.ClientSession() as session:

async with session.get(request\_url, headers=self.headers, ssl=False) as response:

return await response.json()

async def async\_post(self, \*args, \*\*kwargs):

"""

Asynchronous POST request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

headers = {'Content-type': 'application/json'}

headers.update(self.headers)

async with aiohttp.ClientSession() as session:

async with session.post(request\_url, json=self.body\_request, headers=headers, ssl=False) as response:

return await response.json()

async def async\_put(self, \*args, \*\*kwargs):

"""

Asynchronous PUT request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

headers = {'Content-type': 'application/json'}

headers.update(self.headers)

async with aiohttp.ClientSession() as session:

async with session.put(request\_url, json=self.body\_request, headers=headers, ssl=False) as response:

return await response.json()

async def async\_patch(self, \*args, \*\*kwargs):

"""

Asynchronous PATCH request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

headers = {'Content-type': 'application/json'}

headers.update(self.headers)

async with aiohttp.ClientSession() as session:

async with session.patch(request\_url, json=self.body\_request, headers=headers, ssl=False) as response:

return await response.json()

async def async\_delete(self, \*args, \*\*kwargs):

"""

Asynchronous DELETE request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

async with aiohttp.ClientSession() as session:

async with session.delete(request\_url, headers=self.headers, ssl=False) as response:

return await response.json()

def format\_url(self, end\_point):

return self.server + end\_point

@staticmethod

def set\_query\_parameters(request\_url, query\_parameters):

request\_url += '?'

for query\_parameter in query\_parameters:

request\_url += '{}={}&'.format(query\_parameter, query\_parameters[query\_parameter])

return request\_url

kasik1: you can implment ayncy api here

GitHub Copilot: To implement asynchronous API methods in the `BaseEndPoint` class, you can use Python's `asyncio` and `aiohttp` libraries. Below is an example of how you can refactor the `BaseEndPoint` class to include asynchronous versions of the `get`, `post`, `put`, `patch`, and `delete` methods.

```python

import aiohttp

import asyncio

import urllib3

urllib3.disable\_warnings(category=urllib3.exceptions.InsecureRequestWarning)

class BaseEndPoint:

def \_\_init\_\_(self, session, server\_url, endpoint=''):

"""

:param session: Fixture that will authenticate user's credentials

:param server\_url: Server on which the tests will execute

:param endpoint: API endpoint

"""

self.session = session

self.server = server\_url

self.endpoint = endpoint

self.headers = {}

self.body\_request = None

self.query\_parameters = None

self.path\_parameters = None

self.attachment\_file\_path = None

self.attachment\_file\_name = None

async def async\_get(self, \*args, \*\*kwargs):

"""

Asynchronous GET request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

if kwargs:

request\_url = self.set\_query\_parameters(request\_url, kwargs)

async with aiohttp.ClientSession() as session:

async with session.get(request\_url, headers=self.headers, ssl=False) as response:

return await response.json()

async def async\_post(self, \*args, \*\*kwargs):

"""

Asynchronous POST request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

headers = {'Content-type': 'application/json'}

headers.update(self.headers)

async with aiohttp.ClientSession() as session:

async with session.post(request\_url, json=self.body\_request, headers=headers, ssl=False) as response:

return await response.json()

async def async\_put(self, \*args, \*\*kwargs):

"""

Asynchronous PUT request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

headers = {'Content-type': 'application/json'}

headers.update(self.headers)

async with aiohttp.ClientSession() as session:

async with session.put(request\_url, json=self.body\_request, headers=headers, ssl=False) as response:

return await response.json()

async def async\_patch(self, \*args, \*\*kwargs):

"""

Asynchronous PATCH request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

headers = {'Content-type': 'application/json'}

headers.update(self.headers)

async with aiohttp.ClientSession() as session:

async with session.patch(request\_url, json=self.body\_request, headers=headers, ssl=False) as response:

return await response.json()

async def async\_delete(self, \*args, \*\*kwargs):

"""

Asynchronous DELETE request

:param args: Contains the path parameters

:param kwargs: Contains the query parameters

:return: Request response

"""

request\_url = self.format\_url(self.endpoint).format(\*args)

async with aiohttp.ClientSession() as session:

async with session.delete(request\_url, headers=self.headers, ssl=False) as response:

return await response.json()

def format\_url(self, end\_point):

return self.server + end\_point

@staticmethod

def set\_query\_parameters(request\_url, query\_parameters):

request\_url += '?'

for query\_parameter in query\_parameters:

request\_url += '{}={}&'.format(query\_parameter, query\_parameters[query\_parameter])

return request\_url

```

### Key Changes:

1. \*\*Added `aiohttp` for asynchronous HTTP requests.\*\*

2. \*\*Implemented `async\_get`, `async\_post`, `async\_put`, `async\_patch`, and `async\_delete` methods.\*\*

3. \*\*Used `async with` for managing the `aiohttp.ClientSession`.\*\*

4. \*\*Ensured SSL verification is disabled (`ssl=False`) to match the behavior of `requests` with `verify=False`.\*\*

### Usage Example:

```python

import asyncio

async def main():

endpoint = BaseEndPoint(None, "https://example.com", "/api/resource")

response = await endpoint.async\_get()

print(response)

asyncio.run(main())

```

This approach allows you to perform non-blocking API calls, which is useful for high-performance applications.

kasik1: way callling it will change or smae

class Circuits(BaseEndPoint):

def \_\_init\_\_(self, session, server\_url, \*\*kwargs):

super(Circuits, self).\_\_init\_\_(session, server\_url, \*\*kwargs)

GitHub Copilot: The way you call the `Circuits` class will remain the same. The `Circuits` class inherits from `BaseEndPoint`, and its `\_\_init\_\_` method simply passes the arguments to the parent class using `super()`. Whether you use synchronous or asynchronous methods in `BaseEndPoint`, the instantiation of the `Circuits` class does not change.

### Example Usage (No Change):

```python

# Instantiate the Circuits class

circuits = Circuits(session, "https://example.com", endpoint="/api/circuits")

# Call synchronous methods

response = circuits.get()

print(response)

# Call asynchronous methods

import asyncio

async def main():

response = await circuits.async\_get()

print(response)

asyncio.run(main())

```

### Key Points:

- The `Circuits` class is a thin wrapper around `BaseEndPoint`, so its behavior depends entirely on the implementation of `BaseEndPoint`.

- If you add asynchronous methods to `BaseEndPoint`, you can directly use them with `Circuits` without modifying the `Circuits` class itself.

kasik1: just read fixures share draw backs still sync api calls its the best and insted of excel reusablites osn we can use

GitHub Copilot: ### Drawbacks of Current Fixtures with Synchronous API Calls

1. \*\*Blocking Nature of Synchronous Calls\*\*:

- The current implementation uses synchronous API calls (`requests` library). This can lead to performance bottlenecks, especially when dealing with multiple API calls or slow network responses. Each call blocks the execution until it completes, which is inefficient for high-throughput scenarios.

2. \*\*Limited Scalability\*\*:

- Synchronous calls are not ideal for scenarios where multiple test cases need to run concurrently. This can slow down test execution significantly, especially in CI/CD pipelines.

3. \*\*Tight Coupling with Excel\*\*:

- The `test\_data` fixture relies on an `excel` object to fetch test data. This approach is tightly coupled with Excel files, making it less reusable and harder to adapt to other data sources like databases, JSON, or CSV files.

4. \*\*Error Handling\*\*:

- The current implementation lacks robust error handling for API calls. For example, if an API call fails, there is no retry mechanism or detailed logging to help debug the issue.

5. \*\*Hardcoded Assumptions\*\*:

- The `test\_data` fixture assumes specific column names like `TEST\_TYPE` and `TEST\_CASE\_ID`. This makes the fixture less flexible and harder to adapt to different test data formats.

---

### Suggested Improvements

#### 1. \*\*Switch to Asynchronous API Calls\*\*

- Replace the synchronous `requests` library with `aiohttp` for non-blocking, asynchronous API calls. This will improve performance and scalability, especially for tests that involve multiple API calls.

Example:

```python

import aiohttp

import pytest

import asyncio

@pytest.fixture

async def async\_session():

async with aiohttp.ClientSession() as session:

yield session

```

Usage in tests:

```python

async def test\_api\_call(async\_session):

async with async\_session.get("https://example.com/api") as response:

assert response.status == 200

```

#### 2. \*\*Replace Excel with CSV or JSON for Reusability\*\*

- Instead of relying on Excel files, use CSV or JSON files for test data. These formats are easier to parse, more lightweight, and widely supported.

Example with CSV:

```python

import csv

import pytest

@pytest.fixture

def test\_data\_from\_csv():

with open("test\_data.csv", mode="r") as file:

reader = csv.DictReader(file)

data = [row for row in reader]

return data

```

Example with JSON:

```python

import json

import pytest

@pytest.fixture

def test\_data\_from\_json():

with open("test\_data.json", mode="r") as file:

data = json.load(file)

return data

```

#### 3. \*\*Add Retry Mechanism for API Calls\*\*

- Implement a retry mechanism for API calls to handle transient failures.

Example:

```python

import requests

from requests.adapters import HTTPAdapter

from urllib3.util.retry import Retry

def get\_session\_with\_retries():

session = requests.Session()

retries = Retry(total=3, backoff\_factor=1, status\_forcelist=[500, 502, 503, 504])

session.mount("http://", HTTPAdapter(max\_retries=retries))

session.mount("https://", HTTPAdapter(max\_retries=retries))

return session

```

#### 4. \*\*Parameterize Fixtures for Flexibility\*\*

- Use parameterized fixtures to make them more reusable across different test cases.

Example:

```python

@pytest.fixture

def test\_data(request):

data\_source = request.param

if data\_source == "csv":

# Load data from CSV

pass

elif data\_source == "json":

# Load data from JSON

pass

return data

```

#### 5. \*\*Improve Logging and Debugging\*\*

- Enhance logging to capture detailed information about API calls and test data processing.

Example:

```python

import logging

logger = logging.getLogger("test\_logger")

logger.setLevel(logging.DEBUG)

@pytest.fixture

def log\_api\_call():

def \_log(response):

logger.debug(f"URL: {response.url}")

logger.debug(f"Status Code: {response.status\_code}")

logger.debug(f"Response: {response.text}")

return \_log

```

---

### Why These Changes Are Better

1. \*\*Asynchronous API Calls\*\*:

- Non-blocking calls improve performance and allow concurrent execution of tests.

2. \*\*CSV/JSON for Test Data\*\*:

- These formats are lightweight, easier to parse, and more reusable across different test environments.

3. \*\*Retry Mechanism\*\*:

- Reduces test flakiness caused by transient network or server issues.

4. \*\*Parameterization\*\*:

- Makes fixtures more flexible and adaptable to different test scenarios.

5. \*\*Enhanced Logging\*\*:

- Provides better insights into test failures, making debugging easier.

By implementing these changes, the test suite will become more efficient, scalable, and maintainable.

Similar code found with 1 license type

kasik1: we are centraised json for login , for pytet html reprting ,headlesss we have implemnted

GitHub Copilot: It seems you have already implemented centralized JSON for login credentials, pytest HTML reporting, and headless browser configurations. Below is a summary of the benefits and potential improvements for these implementations:

---

### \*\*1. Centralized JSON for Login\*\*

#### \*\*Benefits:\*\*

- \*\*Single Source of Truth\*\*: Centralizing login credentials in a JSON file ensures consistency and reduces duplication.

- \*\*Ease of Maintenance\*\*: Updating credentials in one place simplifies management.

- \*\*Security\*\*: JSON files can be encrypted or stored securely, reducing the risk of exposing sensitive data.

- \*\*Flexibility\*\*: JSON is a lightweight and widely supported format, making it easy to integrate with other systems.

#### \*\*Potential Improvements:\*\*

- \*\*Environment-Specific Credentials\*\*: Use environment variables or separate JSON files for different environments (e.g., development, staging, production).

- \*\*Encryption\*\*: Encrypt the JSON file or use a secrets management tool (e.g., AWS Secrets Manager, HashiCorp Vault) to enhance security.

- \*\*Dynamic Loading\*\*: Dynamically load credentials based on the environment or test configuration.

---

### \*\*2. Pytest HTML Reporting\*\*

#### \*\*Benefits:\*\*

- \*\*Enhanced Test Visibility\*\*: HTML reports provide a clear and detailed view of test results, including pass/fail status, execution time, and logs.

- \*\*Custom Styling\*\*: The use of custom styles (e.g., `style`, `modal`) makes the reports visually appealing and easier to navigate.

- \*\*Screenshots for UI Tests\*\*: Capturing screenshots for failed UI tests helps in debugging issues quickly.

- \*\*Integration with CI/CD\*\*: HTML reports can be easily integrated into CI/CD pipelines for test result visualization.

#### \*\*Potential Improvements:\*\*

- \*\*Add More Metadata\*\*: Include additional metadata in the report, such as test environment details, browser versions, and API endpoints.

- \*\*Interactive Reports\*\*: Add collapsible sections or filters to make large reports easier to navigate.

- \*\*Export Options\*\*: Provide options to export reports in other formats (e.g., PDF, JSON) for sharing or archiving.

- \*\*Error Tracebacks\*\*: Include detailed error tracebacks in the report for better debugging.

---

### \*\*3. Headless Browser Configuration\*\*

#### \*\*Benefits:\*\*

- \*\*Faster Execution\*\*: Headless mode reduces overhead by skipping the rendering of the browser UI, leading to faster test execution.

- \*\*CI/CD Compatibility\*\*: Headless browsers are ideal for running tests in CI/CD pipelines where no GUI is available.

- \*\*Resource Efficiency\*\*: Consumes fewer system resources compared to running a full browser instance.

#### \*\*Potential Improvements:\*\*

- \*\*Dynamic Configuration\*\*: Allow switching between headless and non-headless modes via command-line arguments or environment variables.

- \*\*Browser Logs\*\*: Capture browser logs (e.g., console errors, network requests) for debugging purposes.

- \*\*Parallel Execution\*\*: Combine headless mode with parallel test execution to further speed up test runs.

- \*\*Cross-Browser Testing\*\*: Ensure headless configurations are available for multiple browsers (e.g., Chrome, Firefox).

---

### \*\*Additional Suggestions\*\*

1. \*\*Centralized Test Data Management\*\*:

- Use a centralized JSON or YAML file for all test data, not just login credentials. This improves reusability and consistency across tests.

2. \*\*Retry Mechanism for Flaky Tests\*\*:

- Implement a retry mechanism for tests that fail due to transient issues (e.g., network delays, API timeouts).

3. \*\*Asynchronous API Calls\*\*:

- If your tests involve API calls, consider switching to asynchronous calls (e.g., using `aiohttp`) to improve performance.

4. \*\*Enhanced Logging\*\*:

- Add structured logging (e.g., JSON logs) to capture detailed information about test execution, making it easier to analyze failures.

5. \*\*Test Coverage Reports\*\*:

- Integrate tools like `pytest-cov` to generate test coverage reports and identify untested code paths.

6. \*\*Secrets Management\*\*:

- Replace hardcoded credentials with a secrets management solution to enhance security.

---

By implementing these improvements, you can further enhance the efficiency, maintainability, and security of your test framework. Let me know if you'd like help with any specific implementation!

kasik1: yes all

GitHub Copilot: ### Centralized JSON for Login Credentials

The django\_api\_config.py file demonstrates a centralized approach to managing credentials using a secrets management library (`ndosecrets`). This is a good practice as it abstracts sensitive information and allows dynamic retrieval of credentials based on the environment.

---

### \*\*Benefits of Current Implementation\*\*

1. \*\*Dynamic Environment Handling\*\*:

- The `NAUTOBOT\_ENVIRONMENT` variable dynamically determines the environment (e.g., development, staging, production) and retrieves the corresponding credentials.

2. \*\*Secrets Management\*\*:

- Using `get\_secret` and `get\_local\_secret` ensures that credentials are securely stored and retrieved, reducing the risk of hardcoding sensitive information.

3. \*\*Reusability\*\*:

- The `get\_project\_credentials` function provides a reusable way to fetch credentials for different projects (e.g., `DJANGO`).

4. \*\*Error Handling\*\*:

- The `try-except` block ensures that errors in credential retrieval are caught and handled gracefully.

---

### \*\*Drawbacks and Improvements\*\*

1. \*\*Limited Secrets Integration\*\*:

- The secrets library (`ndosecrets`) is not fully integrated, as noted in the comment: `"work around until the secrets library is fully integrated"`. Completing this integration would enhance security and maintainability.

2. \*\*Hardcoded Project Name\*\*:

- The default project name (`DJANGO`) is hardcoded in `get\_project\_credentials`. This could be parameterized further to support multiple projects dynamically.

3. \*\*Error Logging\*\*:

- The `except` block raises a generic exception without logging the error details. Adding logging would help in debugging issues.

4. \*\*Environment-Specific Logic\*\*:

- The environment is determined using `get\_local\_secret("environment")`. This could be enhanced by allowing overrides via environment variables or command-line arguments for more flexibility.

---

### \*\*Proposed Enhancements\*\*

#### 1. \*\*Complete Secrets Integration\*\*

- Fully integrate the secrets library to avoid the need for workarounds.

#### 2. \*\*Add Logging for Errors\*\*

```python

import logging

logger = logging.getLogger(\_\_name\_\_)

def get\_project\_credentials(project\_name='DJANGO'):

try:

username = VAULT\_SETTINGS.get("testing\_username")

password = VAULT\_SETTINGS.get("testing\_password")

return username, password

except Exception as e:

logger.error(f"Error retrieving credentials for project {project\_name}: {e}")

raise

```

#### 3. \*\*Environment Overrides\*\*

- Allow the environment to be overridden via environment variables or command-line arguments:

```python

import os

NAUTOBOT\_ENVIRONMENT = os.getenv("NAUTOBOT\_ENVIRONMENT", get\_local\_secret("environment"))

```

#### 4. \*\*Support Multiple Projects\*\*

- Extend `get\_project\_credentials` to support multiple projects dynamically:

```python

def get\_project\_credentials(project\_name):

try:

project\_settings = VAULT\_SETTINGS.get(project\_name.lower())

username = project\_settings.get("username")

password = project\_settings.get("password")

return username, password

except Exception as e:

logger.error(f"Error retrieving credentials for project {project\_name}: {e}")

raise

```

---

### \*\*Headless Browser Configuration\*\*

The `driver\_config.py` file already includes headless browser support (though incomplete). Here's how you can finalize it:

#### \*\*Enhancements for Headless Mode\*\*

1. \*\*Complete Headless Mode Implementation\*\*:

```python

if ui\_mode == "headless":

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

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

```

2. \*\*Dynamic Configuration\*\*:

- Allow switching between headless and non-headless modes via command-line arguments:

```python

@staticmethod

def \_get\_chrome\_options(ui\_mode=None):

options = ChromeOptions()

if ui\_mode == "headless":

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

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

return options

```

3. \*\*Cross-Browser Support\*\*:

- Ensure similar configurations for Firefox:

```python

@staticmethod

def \_get\_firefox\_driver(ui\_mode=None):

options = FirefoxOptions()

if ui\_mode == "headless":

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

return webdriver.Firefox(executable\_path=FIREFOXDRIVER\_PATH, options=options)

```

---

### \*\*Pytest HTML Reporting\*\*

The `test\_reporting.py` file already includes hooks for pytest HTML reporting. Here are some additional improvements:

#### \*\*Enhancements for HTML Reporting\*\*

1. \*\*Add Metadata to Reports\*\*:

- Include environment details, browser, and test mode in the report:

```python

def pytest\_html\_results\_summary(prefix, summary, postfix):

prefix.extend([html.p(f"Environment: {NAUTOBOT\_ENVIRONMENT}"),

html.p(f"Browser: {browser}"),

html.p(f"Test Mode: {test\_mode}")])

```

2. \*\*Attach Screenshots for Failures\*\*:

- Ensure screenshots are attached for failed tests:

```python

def attach\_screenshot(item, report, extra, pytest\_html):

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

screenshot\_path = f"screenshots/{item.nodeid.replace('::', '\_')}.png"

driver.save\_screenshot(screenshot\_path)

extra.append(pytest\_html.extras.image(screenshot\_path))

```

3. \*\*Interactive Reports\*\*:

- Add collapsible sections for logs and tracebacks to improve readability.

---

### \*\*Centralized JSON for Login\*\*

If you want to replace the CSV-based login (`users.py`) with a JSON-based approach, here’s how you can do it:

#### \*\*JSON File Example\*\*

```json

{

"users": [

{

"username": "user\_base\_ssot\_admin",

"password": "ssotadmin",

"user\_group": "G\_SSOT\_APP\_ADMIN"

},

{

"username": "user\_base\_ssot\_sites",

"password": "ssotsites",

"user\_group": "G\_SSOT\_APP\_ADMIN"

}

]

}

```

#### \*\*Updated `users.py`\*\*

```python

import json

import os

def users():

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

file\_path = os.path.join(path, 'users.json')

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

return json.load(file)['users']

```

---

### \*\*Summary\*\*

1. \*\*Centralized JSON for Login\*\*:

- Replace CSV with JSON for better flexibility and security.

- Use secrets management for sensitive data.

2. \*\*Headless Browser\*\*:

- Finalize headless mode implementation and add cross-browser support.

3. \*\*Pytest HTML Reporting\*\*:

- Enhance reports with metadata, screenshots, and interactive elements.

4. \*\*Dynamic Environment Handling\*\*:

- Allow environment overrides via environment variables or command-line arguments.

These changes will make your framework more robust, scalable, and maintainable. Let me know if you'd like help implementing any specific part!