API testing

Introduction

Playwright can be used to get access to the REST API of your application.

Sometimes you may want to send requests to the server directly from Python without loading a page and running js code in it. A few examples where it may come in handy:

- Test your server API.
- Prepare server side state before visiting the web application in a test.
- Validate server side post-conditions after running some actions in the browser.

All of that could be achieved via APIRequestContext methods.

The following examples rely on the `pytest-playwright` package which add Playwright fixtures to the Pytest test-runner.

Writing API Test

APIRequestContext can send all kinds of HTTP(S) requests over network.

The following example demonstrates how to use Playwright to test issues creation via GitHub API. The test suite will do the following:

- Create a new repository before running tests.
- Create a few issues and validate server state.
- Delete the repository after running tests.

Configure

GitHub API requires authorization, so we'll configure the token once for all tests. While at it, we'll also set the `baseURL` to simplify the tests.

```
import os
from typing import Generator
```

```
import pytest
from playwright.sync_api import Playwright, APIRequestContext

GITHUB_API_TOKEN = os.getenv("GITHUB_API_TOKEN")
assert GITHUB_API_TOKEN, "GITHUB_API_TOKEN is not set"


@pytest.fixture(scope="session")
def api_request_context(
    playwright: Playwright,
) -> Generator[APIRequestContext, None, None]:
    headers = {
        # We set this header per GitHub guidelines.
        "Accept": "application/vnd.github.v3+json",
        # Add authorization token to all requests.
        # Assuming personal access token available in the environment.
        "Authorization": f"token {GITHUB_API_TOKEN}",
    }
    request_context = playwright.request.new_context(
        base_url="https://api.github.com", extra_http_headers=headers
    )
    yield request_context
    request_context.dispose()
```

Write tests

Now that we initialized request object we can add a few tests that will create new issues in the repository.

```
import os
from typing import Generator

import pytest
from playwright.sync_api import Playwright, APIRequestContext

GITHUB_API_TOKEN = os.getenv("GITHUB_API_TOKEN")
assert GITHUB_API_TOKEN, "GITHUB_API_TOKEN is not set"

GITHUB_USER = os.getenv("GITHUB_USER")
assert GITHUB_USER, "GITHUB_USER is not set"

GITHUB_REPO = "test"
```

```
# ...

def test_should_create_bug_report(api_request_context: APIRequestContext) -> None:
    data = {
        "title": "[Bug] report 1",
        "body": "Bug description",
    }
    new_issue =
api_request_context.post(f"/repos/{GITHUB_USER}/{GITHUB_REPO}/issues", data=data)
    assert new_issue.ok

    issues = api_request_context.get(f"/repos/{GITHUB_USER}/{GITHUB_REPO}/issues")
    assert issues.ok
    issues_response = issues.json()
    issue = list(filter(lambda issue: issue["title"] == "[Bug] report 1",
issues_response))[0]
    assert issue
    assert issue["body"] == "Bug description"

def test_should_create_feature_request(api_request_context: APIRequestContext) ->
None:
    data = {
        "title": "[Feature] request 1",
        "body": "Feature description",
    }
    new_issue =
api_request_context.post(f"/repos/{GITHUB_USER}/{GITHUB_REPO}/issues", data=data)
    assert new_issue.ok

    issues = api_request_context.get(f"/repos/{GITHUB_USER}/{GITHUB_REPO}/issues")
    assert issues.ok
    issues_response = issues.json()
    issue = list(filter(lambda issue: issue["title"] == "[Feature] request 1",
issues_response))[0]
    assert issue
    assert issue["body"] == "Feature description"
```

Setup and teardown

These tests assume that repository exists. You probably want to create a new one before running tests and delete it afterwards. Use a session fixture for that. The part before `yield` is the before all and after is the after all.

```
# ...
@pytest.fixture(scope="session", autouse=True)
```

```
def create_test_repository(
    api_request_context: APIRequestContext,
) -> Generator[None, None, None]:
    # Before all
    new_repo = api_request_context.post("/user/repos", data={"name": GITHUB_REPO})
    assert new_repo.ok
    yield
    # After all
    deleted_repo =
api_request_context.delete(f"/repos/{GITHUB_USER}/{GITHUB_REPO}")
    assert deleted_repo.ok
```

Complete test example

Here is the complete example of an API test:

```
from enum import auto
import os
from typing import Generator

import pytest
from playwright.sync_api import Playwright, Page, APIRequestContext, expect

GITHUB_API_TOKEN = os.getenv("GITHUB_API_TOKEN")
assert GITHUB_API_TOKEN, "GITHUB_API_TOKEN is not set"

GITHUB_USER = os.getenv("GITHUB_USER")
assert GITHUB_USER, "GITHUB_USER is not set"

GITHUB_REPO = "test"


@pytest.fixture(scope="session")
def api_request_context(
    playwright: Playwright,
) -> Generator[APIRequestContext, None, None]:
    headers = {
        # We set this header per GitHub guidelines.
        "Accept": "application/vnd.github.v3+json",
        # Add authorization token to all requests.
        # Assuming personal access token available in the environment.
        "Authorization": f"token {GITHUB_API_TOKEN}",
    }
    request_context = playwright.request.new_context(
        base_url="https://api.github.com", extra_http_headers=headers
```

```
    )
    yield request_context
    request_context.dispose()


@pytest.fixture(scope="session", autouse=True)
def create_test_repository(
    api_request_context: APIRequestContext,
) -> Generator[None, None, None]:
    # Before all
    new_repo = api_request_context.post("/user/repos", data={"name": GITHUB_REPO})
    assert new_repo.ok
    yield
    # After all
    deleted_repo =
api_request_context.delete(f"/repos/{GITHUB_USER}/{GITHUB_REPO}")
    assert deleted_repo.ok


def test_should_create_bug_report(api_request_context: APIRequestContext) -> None:
    data = {
        "title": "[Bug] report 1",
        "body": "Bug description",
    }
    new_issue = api_request_context.post(
        f"/repos/{GITHUB_USER}/{GITHUB_REPO}/issues", data=data
    )
    assert new_issue.ok

    issues = api_request_context.get(f"/repos/{GITHUB_USER}/{GITHUB_REPO}/issues")
    assert issues.ok
    issues_response = issues.json()
    issue = list(
        filter(lambda issue: issue["title"] == "[Bug] report 1", issues_response)
    )[0]
    assert issue
    assert issue["body"] == "Bug description"


def test_should_create_feature_request(api_request_context: APIRequestContext) ->
None:
    data = {
        "title": "[Feature] request 1",
        "body": "Feature description",
    }
    new_issue = api_request_context.post(
        f"/repos/{GITHUB_USER}/{GITHUB_REPO}/issues", data=data
```

```
    )
    assert new_issue.ok

    issues = api_request_context.get(f"/repos/{GITHUB_USER}/{GITHUB_REPO}/issues")
    assert issues.ok
    issues_response = issues.json()
    issue = list(
        filter(lambda issue: issue["title"] == "[Feature] request 1",
issues_response)
    )[0]
    assert issue
    assert issue["body"] == "Feature description"
```

Prepare server state via API calls

The following test creates a new issue via API and then navigates to the list of all issues in the project to check that it appears at the top of the list. The check is performed using LocatorAssertions.

```
def test_last_created_issue_should_be_first_in_the_list(api_request_context:
APIRequestContext, page: Page) -> None:
    def create_issue(title: str) -> None:
        data = {
            "title": title,
            "body": "Feature description",
        }
        new_issue = api_request_context.post(
            f"/repos/{GITHUB_USER}/{GITHUB_REPO}/issues", data=data
        )
        assert new_issue.ok
    create_issue("[Feature] request 1")
    create_issue("[Feature] request 2")
    page.goto(f"https://github.com/{GITHUB_USER}/{GITHUB_REPO}/issues")
    first_issue = page.locator("a[data-hovercard-type='issue']").first
    expect(first_issue).to_have_text("[Feature] request 2")
```

Check the server state after running user actions

The following test creates a new issue via user interface in the browser and then checks via API if it was created:

```
def test_last_created_issue_should_be_on_the_server(api_request_context: APIRequestC
page: Page) -> None:
    page.goto(f"https://github.com/{GITHUB_USER}/{GITHUB_REPO}/issues")
    page.locator("text=New issue").click()
    page.locator("[aria-label='Title']").fill("Bug report 1")
    page.locator("[aria-label='Comment body']").fill("Bug description")
    page.locator("text=Submit new issue").click()
    issue_id = page.url.split("/")[-1]

    new_issue =
api_request_context.get(f"https://github.com/{GITHUB_USER}/{GITHUB_REPO}/issues/{issu
    assert new_issue.ok
    assert new_issue.json()["title"] == "[Bug] report 1"
    assert new_issue.json()["body"] == "Bug description"
```

Reuse authentication state

Web apps use cookie-based or token-based authentication, where authenticated state is stored as cookies. Playwright provides api_request_context.storage_state() method that can be used to retrieve storage state from an authenticated context and then create new contexts with that state.

Storage state is interchangeable between BrowserContext and APIRequestContext. You can use it to log in via API calls and then create a new context with cookies already there. The following code snippet retrieves state from an authenticated APIRequestContext and creates a new BrowserContext with that state.

```
request_context = playwright.request.new_context(http_credentials={"username":
"test", "password": "test"})
request_context.get("https://api.example.com/login")
# Save storage state into a variable.
state = request_context.storage_state()

# Create a new context with the saved storage state.
context = browser.new_context(storage_state=state)
```