### **Scenario 1 (UI)**

Create an automated test for the following scenario:

1. Go to https://www.britinsurance.com/
2. Search for term “IFRS 17” in the search bar top right 
3. The expected is 5 titles are returned from the search. Assert on the returned results, for example, “Interim results for the six months ended 30 June 2022”.

**Code:**

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.common.keys import Keys

import time

class TestBritInsuranceSearch:

@pytest.fixture(scope="class")

def setup(self):

driver = webdriver.Chrome()

driver.maximize\_window()

yield driver

driver.quit()

def test\_search\_ifrs17(self, setup):

driver = setup

# Step 1: Go to https://www.britinsurance.com/

driver.get("https://www.britinsurance.com/")

try:

allow\_button = driver.find\_element(By.XPATH, '//\*[@id="CybotCookiebotDialogBodyLevelButtonLevelOptinAllowAll"]')

allow\_button.click()

print("Cookie notification accepted.")

except Exception as e:

print("Cookie notification not found or already accepted:", e)

# Step 2: Search for the term “IFRS 17” in the search bar top right

time.sleep(3)

driver.find\_element(By.CLASS\_NAME, 'component--header\_\_search').click()

search\_bar = driver.find\_element(By.NAME, 'k')

search\_bar.send\_keys("IFRS 17")

# Step 3: Wait for search results to load

time.sleep(5)

# Step 4: Assert that 5 titles are returned from the search

search\_results = driver.find\_element(By.CLASS\_NAME, 'header--search\_\_results').text.split("\n")

assert len(search\_results) == 5, f"Expected 5 search results, but got {len(search\_results)}"

for result in search\_results:

print(result)

print("Test passed. 'IFRS 17' search returned the expected results.")

### **Scenario 2 (API)**

**Test API :** https://restful-api.dev/

1. **Review the API documentation**

### Step 1: Access the API Documentation

1. Visit the homepage of the API: [restful-api.dev](https://restful-api.dev/" \t "_new).
2. Typically, you would find a "Documentation" or "API Reference" link on the homepage that leads you to the API documentation.

### Step 2: Key Sections to Look For in the Documentation

A well-documented API typically provides the following key information:

#### 1. **Overview**

* A high-level introduction to what the API does and how to interact with it.
* Example: The https://restful-api.dev/ API likely provides endpoints for performing CRUD operations (Create, Read, Update, Delete) on resources like "objects."

#### 2. **Authentication**

* Check if the API requires any form of authentication (e.g., API key, OAuth, etc.). This step is critical for securing access to the API.

#### 3. **Endpoints**

* **GET** /objects: Retrieve a list of objects or a specific object by ID.
* **POST** /objects: Create a new object by sending a JSON payload.
* **PATCH** /objects/{id}: Update an existing object by ID (partial update).
* **DELETE** /objects/{id}: Delete an object by ID.

Each endpoint should specify:

* The HTTP method (GET, POST, PATCH, DELETE).
* The endpoint URL.
* Expected request parameters (path, query, or body).
* Example responses (success and error).

#### 4. **Request and Response Formats**

* The expected request format (typically JSON).
* Examples of successful and unsuccessful API responses.
* Any error codes or messages that might be returned by the API.

#### 5. **Rate Limiting and Constraints**

* Information on any rate limits imposed by the API, such as the number of requests allowed per minute.

#### 6. **Common Status Codes**

* 200 OK: Request succeeded.
* 201 Created: New resource successfully created.
* 400 Bad Request: Invalid request data.
* 404 Not Found: Resource with the specified ID not found.
* 500 Internal Server Error: Something went wrong on the server.

#### 7. **Versioning**

* Information about any versions of the API, if applicable, and how to specify which version to use.

#### 8. **Throttling and Rate Limits**

* Check if the API mentions any rate limits to prevent overuse.

1. **Automated Tests: Create automated tests for the PATCH objects/{id} endpoint.**

**Code:**

import pytest

import requests

import json

base\_url = "https://api.restful-api.dev/objects"

headers = {"content-type": "application/json"}

@pytest.fixture

def create\_object():

payload = json.dumps({

"name": "Apple AirPods",

"data": {

"color": "white",

"price": 135

}

})

print("Creating a new object using POST...")

response = requests.post(base\_url, data=payload, headers=headers)

# Verify the POST request is successful

assert response.status\_code == 200, f"Failed to create object. Status code: {response.status\_code}"

print("Object created successfully!")

# Return the created object ID

object\_id = response.json()["id"]

print(f"New object ID: {object\_id}")

return object\_id

def test\_patch\_object(create\_object):

object\_id = create\_object

patch\_payload = json.dumps({

"data": {

"color": "black",

"generation": "3rd",

"price": 1150

}

})

# Send PATCH request

patch\_url = f"{base\_url}/{object\_id}"

print(f"Updating object with ID {object\_id} using PATCH...")

patch\_response = requests.patch(patch\_url, data=patch\_payload, headers=headers)

# Verify the PATCH request is successful

assert patch\_response.status\_code == 200, f"Failed to update object. Status code: {patch\_response.status\_code}"

print("Object updated successfully!")

patched\_data = patch\_response.json()

print(f"Updated object data: {patched\_data}")

assert patched\_data["data"]["color"] == "black", "Color was not updated correctly"

assert patched\_data["data"]["generation"] == "3rd", "Generation was not updated correctly"

assert patched\_data["data"]["price"] == 1150, "Price was not updated correctly"

print("All data verified successfully.")

def test\_patch\_non\_existent\_object():

patch\_payload = json.dumps({

"data": {

"color": "red",

"price": 999

}

})

# Use a non-existent object ID for PATCH request

non\_existent\_id = 99999

patch\_url = f"{base\_url}/{non\_existent\_id}"

print(f"Trying to update non-existent object with ID {non\_existent\_id} using PATCH...")

patch\_response = requests.patch(patch\_url, data=patch\_payload, headers=headers)

# Verify that the PATCH request fails with 404 Not Found

assert patch\_response.status\_code == 404, "Expected 404 for non-existent object"

print(f"Received expected 404 status for non-existent object ID {non\_existent\_id}.")

1. **API Testing Approach: Document a high-level approach for API testing, including potential scenarios for exploratory testing.**

### API Testing Approach

API testing is crucial for ensuring that the application behaves as expected in terms of data retrieval, manipulation, and security. A high-level API testing approach involves various layers of testing to cover functional and non-functional aspects. Below is a structured approach to API testing, along with potential scenarios for exploratory testing.

### 1. **Test Planning & Preparation**

* **Define API Test Cases**: Based on API documentation, identify various API endpoints and create test cases that cover both positive and negative scenarios.
* **Choose Tools**: Select API testing tools such as **Postman**, **RestAssured** (for Java), **Requests** (for Python), **SoapUI**, or **JMeter**.
* **Set Up Testing Environment**: Ensure that the required environment (dev, staging, or test server) is available, and any necessary API keys or authentication tokens are ready.

### 2. **Types of API Tests**

#### A. **Functional Testing**

Functional testing ensures that the API functions correctly and provides the expected responses.

* **Positive Test Cases**:
  + **POST /objects**: Verify that creating a new object with valid data works correctly. Expect a status code of 200 OK or 201 Created.
  + **GET /objects/{id}**: Verify that fetching an object by its ID returns the correct object data and returns a 200 OK status code.
  + **PATCH /objects/{id}**: Verify that updating an object using valid data and an existing ID updates the resource correctly.
  + **DELETE /objects/{id}**: Verify that deleting an object works correctly, ensuring the object is removed from the system.
* **Negative Test Cases**:
  + **POST /objects with Invalid Data**: Submit invalid data (e.g., missing required fields) and ensure the API returns a 400 Bad Request.
  + **GET /objects/{id} with Invalid ID**: Test fetching a non-existent object ID and expect a 404 Not Found.
  + **PATCH /objects/{id} with Invalid Data**: Submit partial updates with invalid or incomplete data, expecting a 400 Bad Request error.
  + **DELETE /objects/{id} Non-Existent Object**: Attempt to delete an object with an invalid ID, expecting a 404 Not Found.

#### B. **Non-Functional Testing**

Non-functional tests ensure that the API meets performance, security, and usability requirements.

* **Performance Testing**:
  + **Load Testing**: Send a high volume of requests (e.g., 1000 requests per minute) to test how the API performs under load.
  + **Stress Testing**: Simulate high traffic or peak load scenarios to observe how the system handles performance degradation and errors.
* **Security Testing**:
  + **Authentication and Authorization**: Test if API endpoints are properly secured (e.g., token-based authentication, OAuth).
  + **Input Validation**: Test the API’s ability to handle malicious inputs (e.g., SQL injection, cross-site scripting).
  + **Sensitive Data Exposure**: Ensure that sensitive information, like passwords or tokens, is not returned in responses or stored insecurely.
* **Rate Limiting**:
  + **Test Rate Limits**: Ensure that rate limiting works as expected. Test how the API handles too many requests in a short period (e.g., 100 requests per minute), and verify that a 429 Too Many Requests error is returned after the limit is exceeded.

#### C. **Boundary and Edge Case Testing**

* **Empty or Null Values**: Test how the API behaves when empty or null values are provided in request payloads.
* **Max Length Fields**: Test fields with the maximum allowed characters.
* **Invalid Data Types**: Test APIs with incorrect data types (e.g., sending a string where a number is expected).
* **Date Formatting**: Test APIs handling dates or time-related data, using valid and invalid date formats.

### 3. **Exploratory Testing Scenarios**

Exploratory testing is more ad-hoc and helps uncover unexpected behaviors or edge cases that are not covered by predefined test cases.

* **Scenario 1: Missing Authentication Token**:
  + Call the PATCH or POST API endpoints without a valid authentication token.
  + Expected result: Should return a 401 Unauthorized status code.
* **Scenario 2: Unusual Data Inputs**:
  + Send extremely large values (e.g., very large text fields or numbers) as inputs in POST or PATCH requests.
  + Expected result: The API should handle the input gracefully without crashing or causing unexpected behavior.
* **Scenario 3: Simultaneous Requests**:
  + Make simultaneous requests (e.g., 10 requests) to the PATCH /objects/{id} endpoint with slightly different payloads for the same object ID.
  + Expected result: The API should handle concurrency correctly (ideally one request is processed at a time, or the object should be locked for updating).
* **Scenario 4: Checking Response Headers**:
  + Check if the API responses contain appropriate headers, such as Content-Type, Cache-Control, Authorization, and Rate-Limit headers.
  + Expected result: Proper headers should be included in the response, especially for rate-limited APIs.
* **Scenario 5: Boundary Testing**:
  + Test the GET /objects endpoint with pagination and limit values to ensure the system handles edge cases (e.g., empty result sets, large lists).
  + Expected result: The API should return correct pagination results, ensuring that no unexpected data is exposed.
* **Scenario 6: Concurrent Data Modifications**:
  + Simulate simultaneous PATCH requests on the same object to see if the API can handle race conditions.
  + Expected result: The API should either apply changes sequentially or use proper version control to prevent data inconsistency.

### 4. **Post-Testing Activities**

**Logging and Monitoring**:

* + Ensure that the API logs all requests and responses for auditing purposes.
  + Monitor the API for any abnormal behavior during the tests.

**Test Reporting**:

* + Generate and share detailed test reports that include:
    - The status of each test (pass/fail).
    - Any bugs or inconsistencies found.
    - Suggestions for improvements or fixes.

**Regression Testing**:

* + After updates or bug fixes, perform regression testing to ensure that previous functionality is not broken.