6. Parsing Complex JSON Response

Parsing

➔ Parsing means "reading and converting data" into a format that is easy to understand and use.

JSON Response Parsing

➔ Reading a JSON response (text) and converting it into something python can understand like a dictionary is called JSON parsing.

➔ Refer to the **complex.json** file. It contains a JSON response representing user data, including user details, recent orders, and metadata.

➔ We will parse this JSON and validate the data using different test cases.

Tasks and Test Scenarios

1. **User Details Validation**
    a. Verify the status field in the JSON response is "success".
    b. Validate the id, name, and email fields of the user.
    c. Confirm the first phone number is of type home with the value 123-456-7890.
    d. Verify the geo coordinates of the user's address are latitude 39.7817 and longitude - 89.6501.
    e. Validate that the user has enabled notifications and is using the "dark" theme.
2. **Recent Orders Validation**
    a. Verify the total number of orders is 2.
    b. Validate the orderId of the first order is 101 and that the total amount is 1226.49.
    c. Confirm the name of the second item in the first order is "Mouse" and its price is 25.50.
    d. Check that the second order has only one item, a "Smartphone", with a price of 799.99.
3. **Preferences and Metadata Validation**
    a. Verify the user speaks three languages: English, Spanish, and French.
    b. Confirm the requestId is "abc123xyz" and the responseTimeMs is less than 300 ms.

Expected Output

1. All assertions should pass successfully:
    a. User details, recent orders, preferences, and metadata will be validated.
    b. Any discrepancies will be highlighted in the test results.

test_parse_complex_json_response_1.py

```
import json, pytest
@pytest.fixture(autouse=True)
def load_json_fixture(request):
  with open("./complex.json","r") as file:
    request.cls.json_response = json.load(file)
class TestParseComplexJsonResponse:
  @pytest.mark.run(order=1)
  def test_user_details_validation(self):
                                    Verify status
    assert self.json_response["status"] == "success", "Expected status to be 'success'"
                                    Validate user details
    user_details = self.json_response["data"]["userDetails"]
```

```
    assert user_details["id"] == 12345, "User ID mismatch"
    assert user_details["name"] == "John Doe", "User name mismatch"
    assert user_details["email"] == "john.doe@example.com", "User email mismatch"
                                        Validate home phone number
    home_phone = user_details["phoneNumbers"][0]
    assert home_phone["number"] == "123-456-7890", "Home phone number mismatch"
    assert home_phone["type"] == "home", "Home phone type mismatch"
                                        Validate geo coordinates
    geo = user_details["address"]["geo"]
    assert geo["latitude"] == 39.7817, "Latitude mismatch"
    assert geo["longitude"] == -89.6501, "Longitude mismatch"
                                        Validate preferences
    preferences = user_details["preferences"]
    assert preferences["notifications"] is True, "Expected notifications to be True"
    assert preferences["theme"] == "dark", "Theme mismatch"
@pytest.mark.run(order=2)
def test_recent_orders_validation(self):
    recent_orders = self.json_response["data"]["recentOrders"]
                                        Verify total number of orders
    assert len(recent_orders) == 2, "Expected 2 recent orders"
                                        Validate first order details
    first_order = recent_orders[0]
    assert first_order["orderId"] == 101, "First order ID mismatch"
    assert first_order["totalAmount"] == 1226.49, "First order total amount mismatch"
                                        Validate second item in the first order
    second_item = first_order["items"][1]
    assert second_item["name"] == "Mouse", "Second item name mismatch in first order"
    assert second_item["price"] == 25.50, "Second item price mismatch in first order"
                                        Validate second order details
    second_order = recent_orders[1]
    assert len(second_order["items"]) == 1, "Second order items count mismatch"
    item = second_order["items"][0]
    assert item["name"] == "Smartphone", "Item name mismatch in second order"
    assert item["price"] == 799.99, "Item price mismatch in second order"
@pytest.mark.run(order=3)
def test_preferences_and_metadata_validation(self):
                                        Validate preferences → languages
    preferences = self.json_response["data"]["userDetails"]["preferences"]
    languages = preferences["languages"]
    assert len(languages) == 3, "Expected 3 languages in preferences"
    assert languages[0] == "English", "First language mismatch"
```

Validate metadata

```
    metadata = self.json_response["meta"]
    assert metadata["requestId"] == "abc123xyz", "Request ID mismatch"
    assert metadata["responseTimeMs"] < 300, "Response time should be less than 300ms"
```

Note

- ➔ In pytest, request is a special pytest fixture object that gives information about the test function or test class where a fixture is being used.
- ➔ request.cls means → "The test class where this fixture is being used".
- ➔ We use this when
  - ◆ We want to share some data (like JSON data from a file) with all test methods inside a class.
  - ◆ So we store that data inside the class itself.

Random User API

- ➔ Now we will use the publicly available Random User API to parse the response and perform validations.
  - ◆ https://randomuser.me/api/?results=3&nat=us
- ➔ It will generate 3 random user profiles from the United States.

test_parse_random_user_api_response_dict.py

```
import pytest
import requests
class TestParseRandomUserResponse:
  API_URL = "https://randomuser.me/api/?results=3&nat=us"
  @pytest.fixture(autouse=True)
  def fetch_users(self):
    response = requests.get(self.API_URL)
    assert response.status_code == 200, f"Expected 200, got {response.status_code}"
    self.response_json = response.json()
    self.users = self.response_json["results"]
  def test_total_users(self):
    assert len(self.users) == 3, "Expected 3 users in the result"
  def test_user_names_exist(self):
    for user in self.users:
      assert user["name"]["first"], "First name should not be empty"
      assert user["name"]["last"], "Last name should not be empty"
  def test_nested_coordinates(self):
    for user in self.users:
      assert "latitude" in user["location"]["coordinates"]
      assert "longitude" in user["location"]["coordinates"]
  def test_email_and_phone(self):
    for user in self.users:
      assert "@" in user["email"], "Invalid email"
      assert user["phone"].strip(), "Phone number should not be empty"
```

```
  def test_picture_urls(self):
      for user in self.users:
          assert user["picture"]["thumbnail"].startswith("http")
  def test_dob_and_age(self):
      for user in self.users:
          assert user["dob"]["age"] > 0
  def test_location_fields(self):
      for user in self.users:
          assert user["location"]["country"] == "United States"
          assert user["location"]["timezone"]["description"]
```

Useful tools to generate and evaluate JSON Paths

1. https://jsonpathfinder.com/
2. https://www.site24x7.com/tools/jsonpath-finder-validator.html
3. https://jsonpath.com/

test_parse_random_user_api_response_jsonpath.py

```
import pytest
import requests
from jsonpath_ng import parse
class TestParseRandomUserResponse:
  API_URL = "https://randomuser.me/api/?results=3&nat=us"
  @pytest.fixture(autouse=True)
  def fetch_users(self):
      response = requests.get(self.API_URL)
      assert response.status_code == 200, f"Expected 200, got {response.status_code}"
      self.response_json = response.json()
  def test_total_users(self):
      total = parse("$.results").find(self.response_json)
      assert len(total[0].value) == 3, "Expected 3 users in the result"
  def test_user_names_exist(self):
      last_names = parse("$.results[*].name.last").find(self.response_json)
      first_names = parse("$.results[*].name.first").find(self.response_json)
      assert all(name.value for name in last_names), "Last names should not be empty"
      assert all(name.value for name in first_names), "First names should not be empty"
  def test_nested_coordinates(self):
      latitudes = parse("$.results[*].location.coordinates.latitude").find(self.response_json)
      longitudes = parse("$.results[*].location.coordinates.longitude").find(self.response_json)
      assert len(latitudes) == 3, "Latitude should be present for all users"
      assert len(longitudes) == 3, "Longitude should be present for all users"
  def test_email_and_phone(self):
      emails = parse("$.results[*].email").find(self.response_json)
```

```
    phones = parse("$.results[*].phone").find(self.response_json)
    assert all("@" in email.value for email in emails), "Invalid email format found"
    assert all(phone.value.strip() != "" for phone in phones), "Phone numbers must not be empty"
  def test_picture_urls(self):
    pictures = parse("$.results[*].picture.thumbnail").find(self.response_json)
    assert all(pic.value.startswith("http") for pic in pictures), "Picture URL must start with http"
  def test_dob_and_age(self):
    ages = parse("$.results[*].dob.age").find(self.response_json)
    assert all(age.value > 0 for age in ages), "Age must be greater than 0"
  def test_location_fields(self):
    countries = parse("$.results[*].location.country").find(self.response_json)
    timezones = parse("$.results[*].location.timezone.description").find(self.response_json)
    assert all(c.value == "United States" for c in countries), "Expected all users from US"
    assert all(tz.value != "" for tz in timezones), "Timezone description should not be empty"
```

Note

- ➔ **parse** is a function from the jsonpath-ng library in Python.
- ➔ The parse() function lets you write a JsonPath expression (like "$.results") and then search through JSON data to extract specific values.
- ➔ The function **parse("path").find(response_json)** returns
  - ◆ A list of matches, where each match is a **DatumInContext** object.
  - ◆ Each **DatumInContext** contains
    - **.value** → actual data from the matched path
    - **.path** → where it was found in the JSON (for reference)
    - **.context** → the surrounding object (rarely needed in testing)

```
[
    DatumInContext(value='Gibson', path=Fields('last'), context=DatumInContext(value={'title': 'Mr'}, path=Fields('name')))
    DatumInContext(value='Ford', path=Fields('last'), context=DatumInContext(value={'title': 'Mr'}, path=Fields('name')))
    DatumInContext(value='Howard', path=Fields('last'), context=DatumInContext(value={'title': 'Mrs'}, path=Fields('name')))
]
```

- ➔ Just ignore everything except **.value** unless we need to debug where it came from.
- ➔ **assert all(...)** check if all items in a list (or iterable) are True.