1. Requests Library Introduction and Http Methods

Requests Library or Requests Module

1. Requests Library is a Python library that is free to use and open-source.
2. It is used for validating and testing APIs.
3. It supports all HTTP methods such as GET, POST, PUT, and DELETE, etc including GraphQL Support.
4. Easily integrates with Unittest, Pytest automation frameworks and CI/CD pipelines.
5. Requests makes handling HTTP requests and responses easy and user-friendly in Python.

Requests module HTTP Methods

1. **GET** → Use to get data from the server.
2. **POST, PUT, and PATCH** → used to send data to the server like when you're adding or updating something.Requires Payload(request body) that contains the data you want to send.
3. **DELETE Request** → Use to delete data from the server.

Pre-requisites for working with Requests library

1. **Python** Programming Language
2. **Pycharm** / VS Code IDE
3. **Pytest** Framework

Environment Setup

➔ Refer **Installation Documents** of Python and Pycharm.

➔ Create a **New Project** in Pycharm → **APITestingRequests**

➔ Create **requirements.txt** file in the project and specify dependencies or libraries of the project.

- requests
- jsonpath-ng
- lxml
- jsonschema
- google-auth
- faker
- pytest
- pytest-order
- pytest-dependency
- **json**

Python Library	Description
requests	HTTP requests library, similar to RestAssured
jsonpath-ng	JSONPath parsing library, similar to json-path
lxml	Library for XML and HTML parsing, similar to xml-path
jsonschema	JSON schema validation library, similar to json-schema-validator
google-auth	Google API authentication with OAuth
faker	Dummy data generation library, similar to Java Faker
pytest	Testing framework, similar to TestNG
pytest-ordering	Control the order of test execution
pytest-dependency	Make dependency between tests
json	Python inbuilt library for handling JSON data or responses

➔ Execute **pip install -r requirements.txt** in Terminal or Command prompt. Upon executing all latest versions of libraries are installed. Check by executing **pip list** in Terminal or Command prompt

Pytest framework

➔ Pytest is a testing framework that supports features like **unit, integration,** and **functional testing** for automating web applications using Selenium and API's using requests module.

Features of pytest

➔ **Fixtures** ⇒ Provides reusable setup and teardown functions with flexible scopes.

➔ **Parallel Testing** ⇒ Supports running tests in parallel using plugins like **pytest-xdist.**

➔ **Skipping Tests** ⇒ Allows tests to be skipped using decorators like **@pytest.mark.skip.**

- ➔ **Ordering Tests** ⇒ Allows tests execution order using decorators like **@pytest.mark.order.**
- ➔ **Grouping Tests** ⇒ Organize and group tests using markers for selective execution.
- ➔ **Parameterization** ⇒ Run the same test with different data using **@pytest.mark.parametrize.**
- ➔ **Detailed Reports** ⇒ Generates detailed test reports with clear pass/fail results.

Project Structure

- ➔ **Project** → **Test suite**(Package/Directory) → **Test cases**(Modules(.py)) → **Test steps**(Test Methods in class)

Naming Conventions in Pytest framework

- ➔ **Modules** names should start with **test_** or ending with **_test**
  - ◆ **test_login.py** or **login_test.py**
- ➔ **Class** names should start with "**Test**"
  - ◆ **TestClass**
- ➔ **Test method** names should start with "**test**"
  - ◆ **testMethod1(self)**

test_1.py

```
import pytest
class TestClass:
  def testMethod1(self):
    print("this is test method1")
  def testMethod2(self):
    print("this is test method2")
```

Ways to execute test case in Pytest in Pycharm IDE

- ➔ Tools → **Python Integrated Tools** → choose pytest as the default test runner.
- ➔ Click on the **green arrow** next to the test function or class. Right-click and choose Run 'pytest for <your test function or class>' from the context menu.
- ➔ **Execute in Terminal -** pytest -v -s package or directory\module name\test method name
  - ◆ **Run single module -** pytest -v -s Day1\test_1.py
  - ◆ **Run all the modules -** pytest -v -s Day1
  - ◆ **Run specific testMethod in module**
    - pytest -v -s Day1\test_1.py::TestClass::testMethod1

test_Ordering.py → pytest-order

```
import pytest
@pytest.mark.order(1)
def test_open_app():
  print("App opened")
@pytest.mark.order(3)
def test_dashboard():
  print("Dashboard loaded")
@pytest.mark.order(2)
def test_login():
  print("Login successful")
```

test_Dependency.py

```
import pytest
@pytest.mark.dependency()
def test_open_app():
  print("App opened")
@pytest.mark.dependency(depends=["test_open_app"])
def test_login():
  print("Login successful")
@pytest.mark.dependency(depends=["test_login"])
def test_dashboard():
  print("Dashboard loaded")
```

Note

- ➔ If we are getting warnings when we execute
  - ◆ Specify markers in the **pytest.ini** file as below. pytest.ini file contains customized markers

```
[pytest]
```

markers=

order

dependency

or

- Execute **pytest -s -v -p no:warnings Day/test_name.py**

Thumb Rules

1. **Pre-conditions** → content type, set cookies, add auth, add param, set headers info etc....
2. **Actions / Steps** → get, post, put, delete
3. **Validations** → Status code, Response Body, Response Time, Content-Type, Headers and Cookies, Content-Type, etc

test_HTTPMethodsDemo.py

```
import requests, pytest
user_id = None
HEADERS = {
  "Content-Type": "application/json",
  "x-api-key": "reqres-free-v1"
}
```

Test to retrieve users and validate the response

```
@pytest.mark.order(1)
def test_get_users():
  res = requests.get("https://reqres.in/api/users?page=2", headers=HEADERS)
  assert res.status_code == 200, "Wrong status"
  assert res.headers["Content-Type"] == "application/json; charset=utf-8", "Wrong content type"
  assert res.elapsed.total_seconds() < 2, "Too slow"
  data = res.json()
  assert data.get("data") is not None, "Data missing"
  print(data)
  assert data.get("page") == 2, "Wrong page"
  assert "email" in res.text, "Email not found"
```

Test to create a new user and validate the response

```
@pytest.mark.dependency()
@pytest.mark.order(2)
def test_create_user():
  global user_id
  payload = {"name": "madhan", "job": "trainer"}
  res = requests.post("https://reqres.in/api/users", json=payload, headers=HEADERS)
  assert res.status_code == 201, "Wrong status"
  assert res.headers["Content-Type"] == "application/json; charset=utf-8", "Wrong content type"
  assert res.elapsed.total_seconds() < 2, "Too slow"
  data = res.json()
  print(data)
```

```
    assert data.get("name") == "madhan", "Name mismatch"
    assert data.get("job") == "trainer", "Job mismatch"
    assert "id" in data, "ID missing"
    user_id = data["id"]
```

Test to update the previously created user

```
@pytest.mark.order(3)
@pytest.mark.dependency(depends=["test_create_user"])
def test_update_user():
    payload = {"name": "kumar", "job": "teacher"}
    res = requests.put(f"https://reqres.in/api/users/{user_id}", json=payload, headers=HEADERS)
    assert res.status_code == 200, "Wrong status"
    assert res.headers["Content-Type"] == "application/json; charset=utf-8", "Wrong content type"
    assert res.elapsed.total_seconds() < 2, "Too slow"
    data = res.json()
    assert data.get("name") == "kumar", "Name mismatch"
    assert data.get("job") == "teacher", "Job mismatch"
    assert "updatedAt" in data, "Missing update info"
    print(data)
```

Test to delete the previously created user

```
@pytest.mark.order(4)
@pytest.mark.dependency(depends=["test_create_user"])
def test_delete_user():
    res = requests.delete(f"https://reqres.in/api/users/{user_id}", headers=HEADERS)
    assert res.headers["Connection"] == "keep-alive", "Wrong content type"
    assert res.elapsed.total_seconds() < 2, "Too slow"
    assert res.status_code == 204, "Wrong status"
    assert res.text == "", "Response not empty"
    print("User deleted successfully.")
```

Command to Execute

➔ Execute **pytest -s -v Day/test_name.py**

Note

➔ **json parameter**

- When we're sending data in JSON format (like a Python dictionary) and want requests to automatically convert it to a JSON string and set the correct Content-Type: application/json header.
- Best for APIs that expect JSON payloads.

```
payload = {"name": "John", "age": 30}
requests.post(url, json=payload)
```

➔ **data parameter**

- When you're sending **form data** or **already serialized** data (like a JSON string or a file).
- Best for form submissions or if you manually handle the JSON conversion.

Form data (like submitting a form)	If you already converted the dictionary to JSON manually

payload = {
"username": "john",
"password": "1234"
}
requests.post(url, data=payload)
	import json
	payload = {"name": "John", "age": 30}
	requests.post(url, data=json.dumps(payload),
	headers={"Content-Type": "application/json"})