✅ Pytest

1.What is requests library?

👉 "requests is a Python library used to send HTTP requests like GET, POST, PUT, and DELETE to interact with APIs easily."

2.How do you make a GET request?

👉 "We use requests.get() to retrieve data from an API and access the response using .status_code and .json()."

3.How do you send data using POST?

👉 "We use requests.post() with data= for form data or json= for JSON payload to submit data to the server."

4.How do you handle headers?

👉 "Headers are passed using the headers parameter to send authentication tokens, content type, or custom metadata."

5.What if request fails or times out?

👉 "We use try-except blocks and handle exceptions like Timeout, HTTPError, or RequestException to prevent test failures."

6.How do you send JSON in POST?

👉 "Use the json= parameter, which automatically serializes the dictionary and sets Content-Type: application/json."

7.How do you read response content?

👉 "We use .json() for JSON response, .text for raw response, .status_code for status validation, and .headers for metadata."

8.How do you send query parameters?

👉 "Use the params= argument, and requests automatically encodes them into the URL."

9.Can you upload files?

👉 "Yes, using the files= parameter in requests.post() with the file opened in binary mode."

10.Real-world use case?

👉 "requests is widely used in API automation testing, data integration, and fetching live API responses in automation frameworks."

11.Difference between is and ==?

👉 "== compares values, while is compares object identity (memory reference)."

12.How does yield work in pytest?

👉 "yield in pytest fixtures separates setup (before yield) and teardown (after yield) for resource management."

11. How does "yield" work in pytest frameworks?
yield splits fixture setup and teardown in resource management.

12. What is Pytest?
A scalable Python testing framework for writing simple and maintainable tests.

13. Why use Pytest for API testing?
Because of simple syntax, fixtures, parameterization, plugins, and reporting.

14. How do you install Pytest?
Using pip install pytest.

15. How do you run Pytest tests?
Using the pytest command.

16. What is an assertion in Pytest?
A statement used to validate expected test results.

17. What is a fixture in Pytest?
Reusable setup/teardown logic shared across tests.

18. How do you use a fixture?
By passing it as a parameter to a test function.

19. What is scope in fixtures?
It controls fixture lifetime (function/class/module/session).

20. What is parameterization?
Running the same test with multiple input datasets.

21. How do you skip tests?
Using @pytest.mark.skip.

22. How do you send a GET request?
Using requests.get() and validating the response.

23. How do you validate JSON response?
By parsing with response.json() and asserting values.

24. How do you send POST request?
Using requests.post() with JSON payload.

25. How do you send headers?
Pass headers dictionary in the request.

26. How do you handle authentication?
Using auth parameters or tokens in headers.

27. How do you validate response time?
By asserting response.elapsed time.

28. How do you validate schema?
Using the jsonschema library.

29. How do you handle dynamic tokens?
Store them in fixtures and reuse across tests.

30. How do you chain API calls?
Extract response data and pass it to the next request.

31. How do you test negative scenarios?
Send invalid inputs and validate error responses.

32. What is conftest.py?
A shared file for global fixtures.

33. What is pytest.ini?
A configuration file for pytest settings.

34. How do you mark tests?
Using @pytest.mark decorators.

35. What is test discovery?
Pytest auto-detects files starting with test_.

36. How do you generate reports?
Using pytest reporting plugins like HTML reports.

37. What is xfail?
Marks expected failing tests.

38. How do you run tests in parallel?
Using pytest-xdist.

39. What is a plugin in Pytest?
An extension that adds extra functionality.

40. How do you capture logs?
Using pytest logging options.

41. What is setup/teardown in Pytest?
Managed using fixtures with yield.

42. How do you structure API framework?
Using modular folders for tests, endpoints, payloads, utils, and configs.

43. What is a base URL fixture?
A centralized reusable API endpoint.

44. How do you use environment configs?
Using config files or environment variables.

45. How do you read test data from JSON?
Using Python’s json module.

46. How do you read CSV test data?
Using Python’s csv module.

47. How do you validate status codes?
Using assertions on response status.

48. How do you reuse API methods?
Through helper or client functions.

49. How do you handle retries?
Using pytest retry plugins.

50. How do you integrate Pytest with CI/CD?
By running tests in pipeline tools like Jenkins or GitHub Actions.

51. How do you test CRUD operations?
Validate create, read, update, and delete workflows.

52. How do you test pagination?
Verify page size and navigation behavior.

53. How do you test rate limits?
Send rapid requests and validate throttling errors.

54. How do you validate error messages?
Assert expected error responses.

55. How do you handle flaky tests?
Stabilize environment and use retries.

56. How do you mock APIs?
Using mocking libraries like pytest-mock.

57. How do you test file uploads?
Using multipart file requests.

58. How do you test API security?
Validate authentication and permissions.

59. How do you generate test data?
Using libraries like Faker.

60. What are best practices in Pytest API testing?
Use modular design, fixtures, data-driven tests, clean assertions, and CI integration.

61.Folder structure

```
pytest-api-framework/
 |
 ├──── tests/
 |     ├──── test_users.py
 |     ├──── test_orders.py
 |     └──── test_auth.py
 |
 ├──── api/
 |     ├──── users_api.py
 |     ├──── orders_api.py
 |     └──── auth_api.py
 |
```

```
├──── utils/
│     ├──── config_reader.py
│     ├──── logger.py
│     ├──── assertions.py
│     └──── helpers.py
│
├──── test_data/
│     ├──── users.json
│     ├──── orders.json
│     └──── test_data.csv
│
├──── configs/
│     ├──── config.yaml
│     ├──── qa.yaml
│     └──── prod.yaml
│
├──── fixtures/
│     └──── common_fixtures.py
│
├──── reports/
│
├──── conftest.py
├──── pytest.ini
├──── requirements.txt
└──── README.md
```

62.Pytest Parallel Execution

Pytest parallel execution allows you to run multiple test cases simultaneously using multiple CPU cores with the help of **pytest-xdist**, which reduces execution time.

◈ Install Required Plugin

pip install pytest-xdist

◈ One-Liner Command to Run in Parallel

pytest -n 4

👉 -n 4 → Runs tests using 4 parallel workers
👉 -n auto → Automatically uses available CPU cores

63.Workers in Pytest (pytest-xdist)
A **worker** in Pytest is a separate process created by the pytest-xdist plugin to execute tests in parallel.

64.Rate limits

In Pytest, rate limit testing is done by sending multiple API requests within a short time and validating that the server returns **HTTP 429 (Too Many Requests)** after exceeding the allowed threshold.

```
import pytest

import requests


@pytest.fixture

def api_client():

    return requests.Session()


def test_rate_limit(api_client):

    responses = [api_client.get(BASE_URL) for _ in range(200)]


    assert any(r.status_code == 429 for r in responses)
```

65.Dynamic Fixture in Pytest — One-Liner Definition + Example

✅ One-Liner Definition:

A **dynamic fixture** is a fixture whose value changes at runtime based on parameters, CLI options, or environment variables.

✅ **Simple Example (Indirect Parametrization – Most Common)**

```
import pytest


@pytest.fixture
def role(request):
    return f"{request.param}_token"


@pytest.mark.parametrize("role", ["admin", "user"], indirect=True)
def test_access(role):
    print(role)
    assert "token" in role
```

66.Decorators in Pytest — One-Liner + Example

✅ **One-Liner Definition:**

A **decorator in pytest** is a function prefixed with @ that modifies or controls the behavior of a test function.

✅ **Example 1 – Skip Decorator**

```
import pytest


@pytest.mark.skip(reason="Feature not ready")
def test_payment():
    assert True
```

👉 This decorator skips the test execution.

✅ **Example 2 – Parametrize Decorator (Most Common)**

```
@pytest.mark.parametrize("num", [1, 2, 3])
def test_numbers(num):
    assert num > 0
```

👉 Runs the test 3 times with different values.

67.Markers in Pytest.

Markers in pytest are decorators used to categorize, control, or modify test execution.

They are written using:

@pytest.mark.marker_name

🔷 Why Markers Are Important?

Markers help you:

- ✅ Group tests (smoke, regression, api, ui)
- ✅ Skip tests
- ✅ Mark expected failures
- ✅ Run selected tests
- ✅ Control CI/CD execution

🔥 1️⃣ Built-in Markers

✅ @pytest.mark.skip

Skips test unconditionally.

```
import pytest


@pytest.mark.skip(reason="Feature not ready")
def test_payment():
    assert True
```

✅ @pytest.mark.skipif

Skips test based on condition.

```
import sys
```

```
import pytest

@pytest.mark.skipif(sys.platform == "win32", reason="Not supported on Windows")
def test_linux_feature():
    assert True
```

✅ @pytest.mark.xfail

Marks test as expected to fail.

```
@pytest.mark.xfail(reason="Known bug")
def test_bug():
    assert 1 == 2
```

🔹 strict xfail

```
@pytest.mark.xfail(strict=True)
def test_bug():
    assert 1 == 2
```

If test passes → becomes failure (XPASS → FAIL)

✅ @pytest.mark.parametrize (Most Important)

Used for data-driven testing.

```
@pytest.mark.parametrize("num", [1, 2, 3])
def test_numbers(num):
    assert num > 0
```

Runs test 3 times.

🔥 2️⃣ Custom Markers (Most Used in Real Projects)

Example:

```
@pytest.mark.smoke
@pytest.mark.api
def test_create_user():
```

```
    assert True
```

✅ Register Markers in pytest.ini (Best Practice)

```
[pytest]
markers =
    smoke: Smoke test cases
    regression: Regression suite
    api: API tests
    ui: UI tests
```

🔥 How to Run Marked Tests

Run only smoke tests:

```
pytest -m smoke
```

Run smoke but not slow:

```
pytest -m "smoke and not slow"
```

Run regression or api:

```
pytest -m "regression or api"
```

🔥 Apply Marker at Different Levels

✅ Function Level

```
@pytest.mark.smoke
def test_login():
    assert True
```

✅ Class Level

```
@pytest.mark.regression
class TestLogin:
```

```
    def test_valid(self):
        assert True
```

✅ Module Level

At top of file:

pytestmark = pytest.mark.api

Applies to all tests in file.

🎯 Real-Time Example (API Automation)

```
@pytest.mark.api
@pytest.mark.smoke
@pytest.mark.parametrize("status", [200, 201, 400])
def test_status_codes(status):
    assert status in [200, 201, 400]
```

🚀 Interview-Level Answer

If interviewer asks:

What are markers in pytest?

👉 Markers are special decorators used to categorize and control test execution like smoke, regression, skip, xfail, and parametrize.

🔥 Quick Summary

Marker	Purpose
skip	Skip test
skipif	Conditional skip
xfail	Expected failure
parametrize	Data-driven testing
custom markers	Grouping & filtering

68.Strict xfail in Pytest (Clear + Interview Ready)

strict=True in **@pytest.mark.xfail** makes the test **FAIL** if it unexpectedly passes (XPASS becomes failure).

69.Pytest Parametrization Using CSV, JSON & Excel (Complete Practical Guide)

Since you're working in **API automation with Pytest**, external test data handling is very important for scalable frameworks.

We'll cover:

1. CSV
2. JSON
3. Excel (XLSX)
4. Real project structure
5. Interview-ready explanation

✅ 1. Parametrization Using CSV

📄 Example: login_data.csv

username,password,expected_status

admin,admin123,200

user,user123,401

🔹 Step 1: Create CSV Reader Utility

```
import csv


def read_csv(file_path):
    with open(file_path, newline='') as file:
        return list(csv.DictReader(file))
```

🔹 Step 2: Use Parametrize

```
import pytest
from utils.data_reader import read_csv

test_data = read_csv("testdata/login_data.csv")

@pytest.mark.parametrize("data", test_data)
def test_login(data):
    assert int(data["expected_status"]) in [200, 401]
```

🔥 **Important**

CSV values are always strings → convert numbers manually.

✅ 2️⃣ Parametrization Using JSON (Best for API)

📄 **Example: login_data.json**

```
[
  {
    "username": "admin",
    "password": "admin123",
    "expected_status": 200
  },
  {
    "username": "user",
    "password": "user123",
    "expected_status": 401
  }
]
```

🧰 **JSON Reader Utility**

```
import json

def read_json(file_path):
    with open(file_path) as file:
        return json.load(file)
```

□ Parametrize with JSON

```
import pytest
from utils.data_reader import read_json

test_data = read_json("testdata/login_data.json")

@pytest.mark.parametrize("data", test_data)
def test_login(data):
    assert data["expected_status"] in [200, 401]
```

🔥 **Why JSON is Better for API?**

- Supports nested payload
- Direct mapping to request body
- Cleaner for complex test data

✅ **3□Parametrization Using Excel (XLSX)**

Excel is common in enterprise projects.

We use openpyxl.

Install:

```
pip install openpyxl
```

📄 **Example: login_data.xlsx**

username	password	expected_status
admin	admin123	200
user	user123	401

□ Excel Reader Utility

```
from openpyxl import load_workbook

def read_excel(file_path, sheet_name):
    workbook = load_workbook(file_path)
    sheet = workbook[sheet_name]

    data = []
    headers = [cell.value for cell in sheet[1]]

    for row in sheet.iter_rows(min_row=2, values_only=True):
        row_data = dict(zip(headers, row))
        data.append(row_data)

    return data
```

□ Parametrize with Excel

```
import pytest
from utils.data_reader import read_excel

test_data = read_excel("testdata/login_data.xlsx", "Sheet1")

@pytest.mark.parametrize("data", test_data)
def test_login(data):
```

```
    assert data["expected_status"] in [200, 401]
```

🔥 4️⃣ Recommended Framework Structure (Professional Level)

```
project/
│
├── tests/
│   └── test_login.py
│
├── testdata/
│   ├── login_data.csv
│   ├── login_data.json
│   ├── login_data.xlsx
│
├── utils/
│   └── data_reader.py
│
└── conftest.py
```

🔥 5️⃣ Advanced: Indirect Parametrization (Processing Data in Fixture)

```
@pytest.fixture
def login_payload(request):
    return {
        "username": request.param["username"],
        "password": request.param["password"]
    }


@pytest.mark.parametrize("login_payload", test_data, indirect=True)
def test_login_api(login_payload):
```

```
    assert login_payload["username"] is not None
```

🎯 Interview-Level Answer

If interviewer asks:

How do you handle external test data in Pytest?

👉 I use @pytest.mark.parametrize with external CSV, JSON, or Excel files.
👉 I create reusable utility functions to read files.
👉 JSON is preferred for API payloads.
👉 I use indirect parametrization when fixtures need to process data.

🔥 Comparison Table

Feature	CSV	JSON	Excel
Simple flat data	✅	✅	✅
Nested payload	❌	✅	❌
Business-friendly	Medium	Low	High
API automation	Medium	Best	Medium

✅ Requests Library – Practical Code Examples

🔹 Basic GET request with validation

```
import requests


def test_get_users():
    url = "https://reqres.in/api/users/2"
    response = requests.get(url, timeout=5)


    assert response.status_code == 200
    data = response.json()
    assert data["data"]["id"] == 2
```

◈ POST request with JSON payload

```
def test_create_user():
    payload = {
        "name": "John",
        "job": "Tester"
    }

    response = requests.post(
        "https://reqres.in/api/users",
        json=payload
    )

    assert response.status_code == 201
```

◈ Sending headers + authentication token

```
def test_auth_api():
    headers = {
        "Authorization": "Bearer my_token",
        "Content-Type": "application/json"
    }

    response = requests.get(
        "https://api.example.com/profile",
        headers=headers
    )

    assert response.status_code == 200
```

◈ Query parameters

```
def test_query_params():
    params = {"page": 2}

    response = requests.get(
        "https://reqres.in/api/users",
        params=params
    )

    assert response.status_code == 200
```

◈ File upload example

```
def test_file_upload():
    with open("test.txt", "rb") as f:
        response = requests.post(
            "https://httpbin.org/post",
            files={"file": f}
        )

    assert response.status_code == 200
```

✅ Pytest Fixtures – Setup & Teardown with yield

◈ Fixture example

```
import pytest
import requests


@pytest.fixture(scope="session")
def base_url():
```

```
    return "https://reqres.in/api"


@pytest.fixture
def api_client(base_url):
    print("Setup")
    yield requests.Session()
    print("Teardown")


def test_users(api_client, base_url):
    response = api_client.get(f"{base_url}/users")
    assert response.status_code == 200
```

👉 yield runs setup **before** the test and teardown **after** the test.

✅ **Parameterization Example**

```
import pytest
import requests


@pytest.mark.parametrize("user_id", [1, 2, 3])
def test_multiple_users(user_id):
    response = requests.get(f"https://reqres.in/api/users/{user_id}")
    assert response.status_code == 200
```

✅ **Schema Validation Example**

```
from jsonschema import validate


schema = {
    "type": "object",
    "properties": {
```

```
        "id": {"type": "integer"}
    },
    "required": ["id"]
}

def test_schema():
    response = requests.get("https://reqres.in/api/users/2")
    data = response.json()["data"]

    validate(instance=data, schema=schema)
```

✅ **Chaining API Calls Example**

```
def test_chain_api():
    # Create user
    create = requests.post(
        "https://reqres.in/api/users",
        json={"name": "Alice"}
    )

    user_id = create.json().get("id")

    # Fetch user using ID
    get_user = requests.get(
        f"https://reqres.in/api/users/{user_id}"
    )

    assert get_user.status_code in [200, 404]
```

✅ **conftest.py Example (Framework Structure)**

```
project/
│
├── tests/
│   └── test_users.py
├── utils/
│   └── api_client.py
├── data/
│   └── test_data.json
└── conftest.py
```

conftest.py

```
import pytest


@pytest.fixture(scope="session")
def config():
    return {"base_url": "https://reqres.in/api"}
```

✅ **Reading JSON Test Data**

```
import json


def load_test_data():
    with open("data/test_data.json") as f:
        return json.load(f)
```

✅ **Running Tests in Parallel**

```
pytest -n 4
(Requires pip install pytest-xdist)
```

✅ **HTML Report Generation**

```
pytest --html=report.html
(Requires pip install pytest-html)
```

✅ Retry Failed Tests

```
pytest --reruns 2
(Requires pip install pytest-rerunfailures)
```

✅ CI/CD Example (GitHub Actions)

```
name: Pytest API Tests

on: [push]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: pip install pytest requests
      - run: pytest
```

✅ Best Practice – API Client Wrapper

Instead of calling requests everywhere:

```
class APIClient:
    def __init__(self, base_url):
        self.base_url = base_url

    def get(self, endpoint):
        return requests.get(f"{self.base_url}/{endpoint}")
```

Usage:

```
def test_users():
    client = APIClient("https://reqres.in/api")
    response = client.get("users")
    assert response.status_code == 200
```
