8. API Chaining using Requests Library

Generate Random Test Data using faker Library

- ➔ Faker is a Python library used to generate fake test data like names, emails, phone numbers, addresses, etc.
- ➔ To install faker package execute
  - ◆ **pip install faker**
- ➔ It is widely used in
  - ◆ Testing APIs, web applications, or forms
  - ◆ Populating databases with dummy data
  - ◆ Creating sample datasets for development

Faker methods in Python

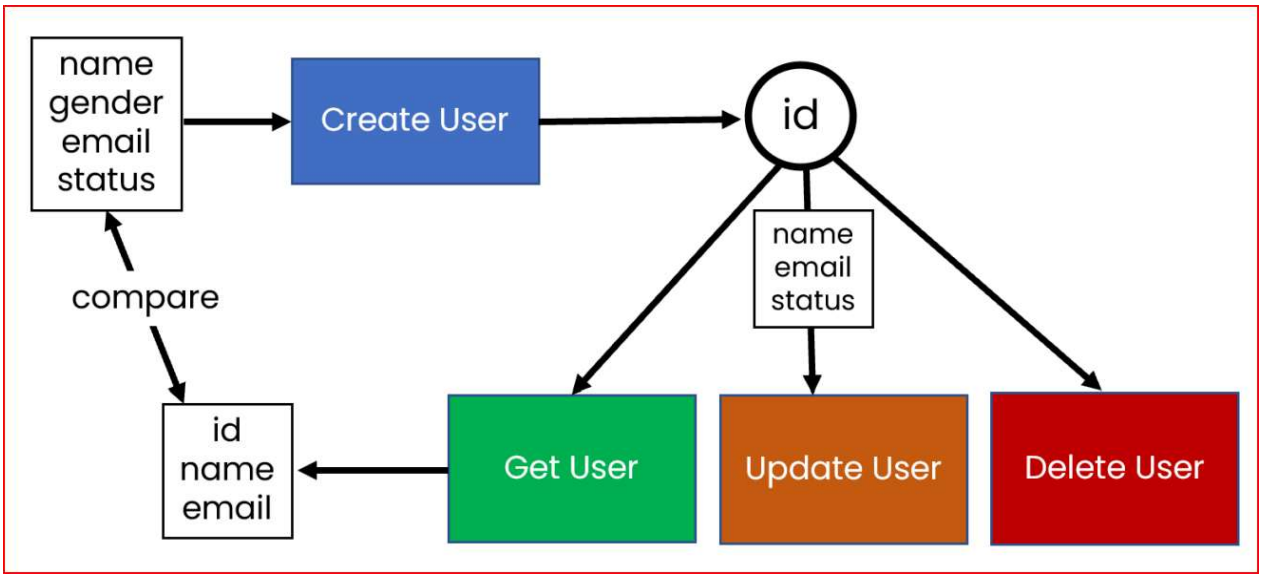
Category	Method Example	Description
Name	faker.name()	Full name
	faker.first_name()	First name
	faker.last_name()	Last name
Email	faker.email()	Random email address
	faker.safe_email()	Safer email (e.g., ends with example.org)
Phone Number	faker.phone_number()	Random phone number
Address	faker.address()	Full address
	faker.city()	Random city
Password	faker.password(length=10)	Random password
Date	faker.date_of_birth()	Random date of birth
Job	faker.job()	Random job title
Company	faker.company()	Random company name

test_generate_fake_data.py

```
from faker import Faker
class TestGenerateTestDataUsingFakerLibrary:
  def test_fake_data_generator(self):
      faker = Faker()
      fullname = faker.name()
      firstname = faker.first_name()
      lastname = faker.last_name()
      email = faker.safe_email()
      password = faker.password(length=8)
      phoneno = faker.phone_number()
      print("Full Name:", fullname)
      print("First Name:", firstname)
      print("Last Name:", lastname)
      print("Email:", email)
      print("Password:", password)
      print("Phone No:", phoneno)
```

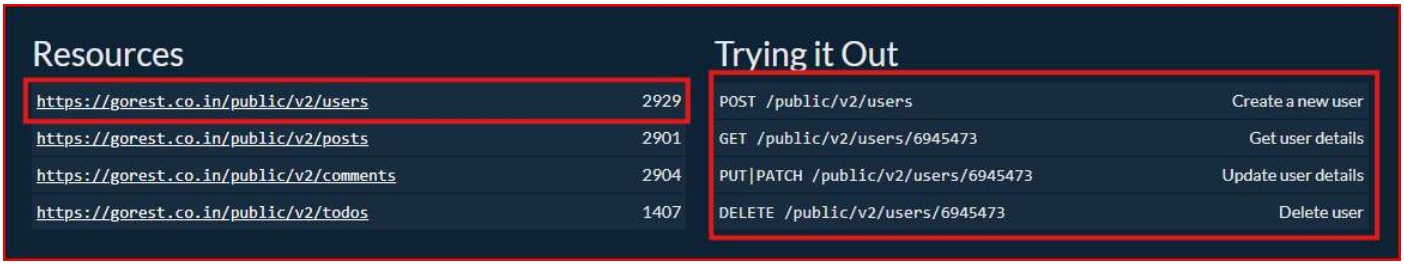
API Chaining

- ➔ API chaining involves executing multiple API requests in sequence, where the response or part of the response from one request is used as input for the next or subsequent requests.
- ➔ If there will be dependencies between multiple API requests in that case we will use API Chaining.
- ➔ We will store Response in variables and we can use those variables as part of the request.



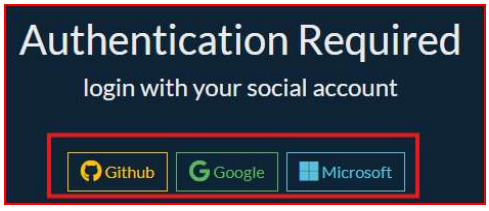
GoRest API

- ➔ GoRest API is a popular public API used for API chaining demonstrations and requires token-based authentication
- ➔ GoRest API Supports GraphQL, RestAPI, etc many more things.
- ➔ Navigate to https://gorest.co.in/ to see API Documentation.
- ➔ We Will use **users Resource** for our demo purpose. It has POST, GET, PUT, DELETE requests.



POST	/public/v2/users	Create a new user
GET	/public/v2/users/23	Get user details
PUT	/public/v2/users/23	Update user details
DELETE	/public/v2/users/23	Delete user

- ➔ Click on **Get your Access Token** to create an Access Token for working with **users resource.**



- ➔ Once we signup navigate to https://gorest.co.in/my-account/access-tokens to see active tokens.

test_chaining_apis.py

```
import pytest
import json
import requests
from faker import Faker
BASE_URL = "https://gorest.co.in/public/v2/users"
TOKEN = "c35e10e748c6f113775527bcef204e9929b4c9f4b995a8ee253eec46aed57b06"
HEADERS = {
  "Authorization": f"Bearer {TOKEN}",
  "Content-Type": "application/json"
}
faker = Faker()
class TestChainingAPIs:
  user_id = None
  @pytest.mark.dependency(name="create")
  def test_create_user(self):
    data = {
      "name": faker.name(),
      "gender": "Male",
      "email": faker.unique.email(),
      "status": "inactive"
    }
    response = requests.post(BASE_URL, json=data, headers=HEADERS)
    assert response.status_code == 201, "Create user failed"
    TestChainingAPIs.user_id = response.json().get("id")
    assert TestChainingAPIs.user_id, "User ID not generated"
    print("\nCREATE Response:\n", json.dumps(response.json(), indent=4))
  @pytest.mark.dependency(depends=["create"])
  def test_get_user(self):
    response = requests.get(f"{BASE_URL}/{TestChainingAPIs.user_id}", headers=HEADERS)
    assert response.status_code == 200, "Get user failed"
    print("\nGET Response:\n", json.dumps(response.json(), indent=4))
  @pytest.mark.dependency(depends=["create"])
  def test_update_user(self):
    updated_data = {
      "name": faker.name(),
      "gender": "Male",
      "email": faker.unique.email(),
      "status": "active"
    }
    response = requests.put(f"{BASE_URL}/{self.user_id}", json=updated_data, headers=HEADERS)
```

```
    assert response.status_code == 200, "Update user failed"
    print("\nPUT Response:\n", json.dumps(response.json(), indent=4))
  @pytest.mark.dependency(depends=["create"])
  def test_delete_user(self):
    response = requests.delete(f"{BASE_URL}/{TestChainingAPIs.user_id}", headers=HEADERS)
    assert response.status_code == 204, "Delete user failed"
```

Note

- ➔ We can access a class variable using self.
- ➔ If an instance variable with the same name exists, it overrides the class variable.
- ➔ If no instance variable exists, self.varname refers to the class variable.

Problem

- ➔ If we create four different classes for four different methods, how can we share the same ID across all the test methods in those classes? That's the main challenge here.

Solutions

- ◆ **Option 1** → Use a Shared Module-Level Variable (Recomended)
- ◆ **Option 2** → Use a Fixture with Scope='session' (Recomended)
- ◆ **Option 3** → Use a Custom Test Base Class with Static Variable (Advanced)

Option 1 → Use a Shared Module-Level Variable

shared_data.py

```
user_id = None
```

test_create_user.py

```
import requests
import json
import pytest
from faker import Faker
import shared_data
BASE_URL = "https://gorest.co.in/public/v2/users"
TOKEN = "c35e10e748c6f113775527bcef204e9929b4c9f4b995a8ee253eec46aed57b06"
HEADERS = {
  "Authorization": f"Bearer {TOKEN}",
  "Content-Type": "application/json"
}
@pytest.mark.run(order=1)
@pytest.mark.dependency(name="TestCreateUser::test_create_user",scope="session")
class TestCreateUser:
  def test_create_user(self):
    faker = Faker()
    data = {
      "name": faker.name(),
      "gender": "Male",
      "email": faker.unique.email(),
```

```
        "status": "inactive"
    }
    response = requests.post(BASE_URL, json=data, headers=HEADERS)
    assert response.status_code == 201, "Create user failed"
    shared_data.user_id = response.json()["id"]
    assert shared_data.user_id is not None, "User ID not saved"
    print("\nCREATE Response:\n", json.dumps(response.json(), indent=4))
```

test_get_user.py

```
import requests
import json
import shared_data
import pytest
BASE_URL = "https://gorest.co.in/public/v2/users"
TOKEN = "c35e10e748c6f113775527bcef204e9929b4c9f4b995a8ee253eec46aed57b06"
HEADERS = {
  "Authorization": f"Bearer {TOKEN}",
  "Content-Type": "application/json"
}
@pytest.mark.run(order=2)
@pytest.mark.dependency(depends=["TestCreateUser::test_create_user"], scope="session")
class TestGetUser:
   def test_get_user(self):
      user_id = shared_data.user_id
      assert user_id is not None, "User ID not set by create user test"
      response = requests.get(f"{BASE_URL}/{user_id}", headers=HEADERS)
      assert response.status_code == 200, "User fetch failed"
      print("\nGET Response:\n", json.dumps(response.json(), indent=4))
```

command to execute

➔ pytest -s -v -p no:warnings

Pros and cons

➔ Simple and readable

➔ Clean separation of test logic

➔ Works only if test_create_user is run first (use pytest-order or dependency markers)

Option 2 → Use a Fixture with Scope='session'

conftest.py

```
import pytest
import json
import requests
from faker import Faker
BASE_URL = "https://gorest.co.in/public/v2/users"
TOKEN = "c35e10e748c6f113775527bcef204e9929b4c9f4b995a8ee253eec46aed57b06"
```

```
HEADERS = {
 "Authorization": f"Bearer {TOKEN}",
 "Content-Type": "application/json"
}
@pytest.fixture(scope="session",autouse=True)
def create_user():
  faker = Faker()
  data = {
     "name": faker.name(),
     "gender": "Male",
     "email": faker.unique.email(),
     "status": "inactive"
  }
  response = requests.post(BASE_URL, json=data, headers=HEADERS)
  assert response.status_code == 201, "Create user failed"
  print("\nCREATE Response:\n", json.dumps(response.json(), indent=4))
  return response.json()["id"]
```

test_get_user.py

```
import requests, json
BASE_URL = "https://gorest.co.in/public/v2/users"
TOKEN = "c35e10e748c6f113775527bcef204e9929b4c9f4b995a8ee253eec46aed57b06"
HEADERS = {
  "Authorization": f"Bearer {TOKEN}",
  "Content-Type": "application/json"
}
class TestGetUser:
   def test_get_user(self,create_user):
      user_id = create_user
      assert user_id is not None, "User ID not set by create user test"
      response = requests.get(f"{BASE_URL}/{user_id}", headers=HEADERS)
      assert response.status_code == 200, "User fetch failed"
      print("\nGET Response:\n", json.dumps(response.json(), indent=4))
```

command to execute

➔ **pytest -s -v -p no:warnings**

Pros

- ➔ Fixture is automatically injected
- ➔ Works across modules (when declared in conftest.py)
- ➔ Cleaner than global variable

Option 3 → Use a Custom Test Base Class with Static Variable (Advanced)

A powerful technique if you're organizing your test suite into multiple classes and want to share state (user_id) cleanly without using external files or fixtures.

➔ **Goals**

- Have 4 different classes (TestCreateUser, TestGetUser, TestUpdateUser, TestDeleteUser)
- Create the user in one class
- Share user_id to all other classes
- Keep everything within test classes (no conftest, no shared module)

Step-by-Step Guide

➔ **Step 1:** Create a Base Class

base_test_user.py

```
class BaseUserTest:
  user_id = None  # shared across all child test classes
```

➔ **Step 2:** Create Separate Test Classes. Each class inherits from BaseUserTest.

test_create_user.py

```
import json
import requests
import pytest
from faker import Faker
from base_test_user import BaseUserTest
BASE_URL = "https://gorest.co.in/public/v2/users"
TOKEN = "c35e10e748c6f113775527bcef204e9929b4c9f4b995a8ee253eec46aed57b06"
HEADERS = {
 "Authorization": f"Bearer {TOKEN}",
 "Content-Type": "application/json"
}
class TestCreateUser(BaseUserTest):
  @pytest.mark.dependency(name="TestCreateUser::test_create_user",scope='session')
  @pytest.mark.run(order=1)
  def test_create_user(self):
     faker = Faker()
     data = {
        "name": faker.name(),
        "gender": "Male",
        "email": faker.unique.email(),
        "status": "inactive"
     }
     response = requests.post(BASE_URL, json=data, headers=HEADERS)
     assert response.status_code == 201, "Create user failed"
     print("\nCREATE Response:\n", json.dumps(response.json(), indent=4))
     BaseUserTest.user_id = response.json()["id"]
     assert BaseUserTest.user_id, "User ID not saved"
```

test_get_user.py

```
import json
```

```
import requests
import pytest
from base_test_user import BaseUserTest
BASE_URL = "https://gorest.co.in/public/v2/users"
TOKEN = "c35e10e748c6f113775527bcef204e9929b4c9f4b995a8ee253eec46aed57b06"
HEADERS = {
 "Authorization": f"Bearer {TOKEN}",
 "Content-Type": "application/json"
}
class TestGetUser(BaseUserTest):
  @pytest.mark.run(order=2)
  @pytest.mark.dependency(depends=["TestCreateUser::test_create_user"],scope="session")
  def test_get_user(self):
      assert BaseUserTest.user_id, "User ID not available"
      response = requests.get(f"{BASE_URL}/{BaseUserTest.user_id}", headers=HEADERS)
      assert response.status_code == 200, "User fetch failed"
      print("\nGET Response:\n", json.dumps(response.json(), indent=4))
```

command to execute

➔ pytest -s -v -p no:warnings