PROJECT 3 TESTING REPORT

Project Name	PYTEST
Section	003
Prepared by	GROUP 1

Team Members

Team Member	Email	Phone
Chenchu Siva Koushik Vemula	ChenchuSivaKoushikVemula@my.unt.edu	9409774833
Jonnala Nihal	nihaljonnala@my.unt.edu	9452741205
Sai Nikitha Ponnathota	SaiNikithaPonnathota@my.unt.edu	4697684866
Teja kamesh Gattu	Tejakameshgattu@my.unt.edu	9409774834
Javali Subha Mapati	JavaliSubhaMapati@my.unt.edu	9408433350
Nikhila Mannem	NikhilaMannem@my.unt.edu	9408433583
Naresh Bakaram	NareshBakaram@my.unt.edu	9409773992

Bhavana Varma Penmetsa	BhavanaVarmaPenmetsa@my.unt.edu	9454004685

Preface

Contents

Overview

A well-liked and effective Python testing framework is Pytest. It has a number of features that help make creating and running tests simpler and more effective. Automatic test discovery, a fixture system, support for parameterized testing, and extensive assertion support are a few of its standout features. A sophisticated command-line interface and a plugin architecture are further features of Pytest that let you expand its capability. For a thorough testing and development workflow, Pytest also interfaces with other tools like coverage and flake8.

You may build clean and succinct tests using Pytest's straightforward and intuitive syntax. Your tests can be set up as Python modules, and Pytest will find and execute them automatically. Fixtures can be used to create and destroy test dependencies like temporary files or database connections. Also supported by Pytest is parameterized testing, which enables you to run the same test with several sets of inputs.

To make writing tests simpler, Pytest includes a variety of assertion tools. These tools offer succinct and unambiguous syntax for typical assertions like membership or equality checks.

In general, Pytest is a thorough and adaptable testing framework that may assist you in creating better tests and enhancing the caliber of your code. It is a preferred option for testing Python programs due to its popularity, versatility, and extensibility.

Features

1. Easy Test Discovery: Pytest can automatically discover and run all the tests in a directory, making it easy to get started with testing.

2. Fixtures: Pytest provides a powerful fixture system that allows you to define and reuse test setup and teardown code.

3. Parametrized Testing: With Pytest, you can run the same test with different sets of inputs using parameterized testing.

4. Advanced Assertion Support: Pytest comes with a range of assertion helpers that make it easy to write clear and concise tests.

5. Command-Line Interface: Pytest has a comprehensive command-line interface that supports a wide range of options for configuring and running tests.

6. Extensibility: Pytest has a plugin architecture that allows you to easily extend its functionality to support new types of tests or reporting formats.

7. Integration with Other Tools: Pytest can integrate with other tools such as coverage and flake8 to provide a complete testing and development workflow.

Pytest is a comprehensive testing framework that provides a range of powerful features for writing and running tests. Its ease of use, flexibility, and extensibility make it a popular choice for testing Python applications.

Roles and Input Combinations

1. Test Runner: Pytest can be used as a test runner to discover and execute tests in a Python project. It supports different types of test functions, such as unit tests, integration tests, and functional tests. Pytest can also run tests in parallel, reducing the time required to execute large test suites.

2. Assertion Library: Pytest provides a rich set of assertion helpers that make it easy to write clear and concise tests. These assertion helpers can be used to write assertions in any Python project, not just in tests.

3. Fixture System: Pytest has a powerful fixture system that allows you to define and reuse test setup and teardown code. This can be used to set up resources required by your tests, such as databases or network connections. Pytest fixtures can also be used to provide test data or mock objects.

4. Parameterized Testing: Pytest supports parameterized testing, allowing you to run the same test with different sets of inputs. This feature can be used to test a function with different inputs, boundary conditions, or edge cases.

5. Plugin Architecture: Pytest has a plugin architecture that allows you to extend its functionality. There are many plugins available that add support for additional testing frameworks, test discovery mechanisms, reporting formats, and more.

6. Integration with Other Tools: Pytest can integrate with other tools like coverage, flake8, and tox to provide a complete testing and development workflow. This can be useful for developers who want to automate their testing and ensure code quality.

Pytest can be used in different roles and input combinations to test different aspects of a Python project. Its versatility and extensibility have made it a popular choice for developers who want to write effective tests and improve the quality of their code.

Economics

1. Time and Cost Savings: Pytest's ease of use, powerful features, and comprehensive command-line interface can help reduce the time and cost of testing. It allows developers to write clear and concise tests that are easy to maintain, reducing the time required for testing and debugging.

2. Code Quality: Pytest's advanced assertion support, parameterized testing, and integration with other tools like coverage and flake8 can help improve code quality. By catching bugs early in the development process, developers can reduce the cost of fixing bugs later on and improve the overall reliability of their code.

3. Extensibility: Pytest's plugin architecture allows developers to extend its functionality to support new types of tests or reporting formats. This can help reduce the cost of developing custom testing tools and allow developers to reuse existing code.

4. Learning Curve: While Pytest is easy to use, there may be a learning curve for developers who are new to the framework. This can impact the economics of using Pytest, as it may require additional training or time to learn the framework.

5. Compatibility: Pytest is compatible with different Python versions, operating systems, and testing frameworks. This can impact the economics of using Pytest, as it may require additional resources to ensure compatibility with existing systems.

Pytest can provide significant economic benefits for testing Python applications, including time and cost savings, improved code quality, and extensibility. However, there may be some challenges to overcome, such as the learning curve and compatibility issues, which can impact the economics of using Pytest.

Open source of PYTEST:

Pytest is an open-source software, which means that its source code is freely available for anyone to view, use, modify, and distribute. This is open-source nature of Pytest provides several benefits:

1. **Accessibility**: Being open source makes Pytest accessible to everyone, regardless of their financial situation or technical expertise. It allows anyone to use and modify the software to suit their needs.

2. **Transparency:** The open-source nature of Pytest provides transparency into the software's workings, making it easier for developers to understand how it works and make improvements or bug fixes.

3. **Collaboration**: The open-source community can collaborate on the development and improvement of Pytest, allowing for a more diverse range of ideas and perspectives.

4. **Innovation**: Open-source software like Pytest can lead to innovation because developers can modify and improve the software to meet their specific needs. These modifications and improvements can then be shared with the wider community, leading to further innovation and improvement.

5. **Reliability**: Because the source code of Pytest is freely available, developers can review and verify it, which leads to a more reliable and trustworthy software.

In summary, the open-source nature of Pytest provides several benefits, including accessibility, transparency, collaboration, innovation, and reliability. These benefits contribute to making Pytest a widely-used and well-regarded testing framework.

Commercial of pytest:

Pytest is a popular testing framework for Python that offers a range of features to help developers write simple and effective tests. Here are some reasons why Pytest might be a good choice for your commercial software development projects:

1. **Easy to use**: Pytest has a simple syntax and provides an intuitive way to write tests, making it easy for developers to get started with testing.

2. **Flexible**: Pytest supports various types of tests including unit tests, integration tests, and functional tests. It also allows developers to create test fixtures to set up preconditions for tests, and to parameterize tests to test multiple scenarios with a single test.

3. **Extensible**: Pytest can be extended with plugins to add new functionality, such as support for testing web applications, database testing, and more.

4. **Comprehensive reporting**: Pytest generates detailed test reports, making it easy to see which tests passed, which failed, and why.

5. **Large community**: Pytest has a large community of developers contributing to it, which means that there is a wealth of knowledge and resources available to help you get started and troubleshoot issues.

Customers Profile:

Pytest is a popular testing framework for Python that can be used by a wide range of customers, including individuals, small businesses, and large enterprises. Here are some examples of potential customers who might benefit from using Pytest:

1. **Individual developers**: Pytest can be used by individual developers who want to write effective tests for their Python code. It offers an easy-to-use syntax and comprehensive reporting, making it easy for developers to test their code and identify any issues.

2. **Small businesses:** Small businesses that develop software in Python can use Pytest to ensure the quality and functionality of their code. Pytest's flexibility and ease of use make it an accessible testing framework for small teams with limited resources.

3. **Startups**: Startups can benefit from using Pytest to quickly and easily test their software products. Pytest can help startups identify and fix issues early in the development process, reducing the risk of product failures and costly rework.

4. **Large enterprises**: Large enterprises that develop complex software products in Python can use Pytest to test their applications at scale. Pytest's advanced features, such as test fixtures and parameterization, make it an ideal testing framework for large and complex codebases.

5. **Software development agencies**: Software development agencies that provide Python development services to clients can use Pytest to ensure the quality of their deliverables.

Pytest can help agencies identify and fix issues before delivering code to clients, improving customer satisfaction and reducing the risk of project delays.

Overall, Pytest can be used by a wide range of customers who develop software in Python and want to ensure the quality and functionality of their code.

History

Pytest was created by Holger Krekel in 2004 as an alternative to the existing Python testing frameworks such as unittest and nose. Krekel wanted to create a testing framework that was easier to use, more flexible, and provided better reporting.

The first version of Pytest was released in 2004 as part of the py library. The framework gained popularity over the years, and in 2009, Pytest was released as a standalone package on PyPI (Python Package Index).

Pytest continued to evolve over the years, with new features being added, such as parameterized testing, fixture system, and advanced assertion support. In 2013, Pytest became compatible with Python 3, which helped increase its popularity among developers who were transitioning to Python 3.

Pytest has been widely adopted by the Python community and is now one of the most popular testing frameworks for Python. Its simplicity, flexibility, and powerful features have made it a favorite among developers who value efficient and effective testing.

In summary, Pytest was created in 2004 by Holger Krekel as an alternative to existing Python testing frameworks. It has since evolved into a popular testing framework known for its simplicity, flexibility, and powerful features.

Testing Interface

Pytest is a powerful testing framework that can be used to test user interfaces (UIs) of web browser applications. In combination with other tools such as Selenium or Playwright, Pytest can be used to automate browser actions and write tests that verify the functionality of the UI.

UI testing with Pytest typically involves setting up a test fixture to create a browser instance, navigating to the application being tested, and then performing actions on the UI to simulate user behavior. Pytest can then be used to assert the expected behavior of the application and ensure that the UI is working as intended.

UI testing with Pytest can help ensure that web browser applications are user-friendly and free from bugs and errors. By automating repetitive and complex browser actions, Pytest makes it easy to test the functionality of the UI across a variety of scenarios, helping to catch bugs early in the development cycle and improve the overall quality of the application.

Testing Methods

Static testing :

Code reviews: Reviewing the source code of a software system to find defects or errors. Example: conducting a code review of a Python module to ensure it follows PEP 8 style guidelines.

Requirements reviews: Reviewing the requirements documents of a software system to ensure they are complete, accurate, and testable. Example: reviewing a software requirements specification document to ensure it includes all necessary features and functions.

Walkthroughs: A group of peers examine code together to identify defects. Example: conducting a walkthrough of a user manual to identify any inconsistencies or inaccuracies in the content.

Dynamic testing :

Unit testing: Testing individual functions or methods within the code of a software system to ensure they produce expected outputs. Example: testing a function that sorts a list of numbers to ensure it sorts the list correctly.

Integration testing: Testing how different modules or components of a software system interact with each other. Example: testing how a database interacts with a web application.

System testing: Testing the entire software system to ensure it meets the functional and non-functional requirements. Example: testing a mobile app's user interface, functionality, and performance to ensure it meets the user's needs and expectations.

Both static testing and dynamic testing are important for ensuring the quality and reliability of a software system. Static testing focuses on the static artifacts of a software system, such as code and requirements documents, while dynamic testing focuses on the dynamic behavior of a software system as it runs.

White-box testing :

White box testing, on the other hand, is a testing approach that examines the internal workings of the software system, including the code, design, and implementation. Testers who use this approach have knowledge of the internal workings of the software system, and they test the system based on its internal structure, architecture, and implementation. This approach is often used to test the software from a developer's perspective and to ensure that it is robust and efficient.

Unit testing: Testing individual functions or methods within the code of a software system. Example: testing a function that calculates the average of a list of numbers.

Integration testing: Testing the interactions between different modules or components of a software system. Example: testing how the front-end and back-end of a web application interact with each other.

Code coverage testing: Testing the amount of code that has been executed by test cases. Example: ensuring that all the code paths of a function have been tested at least once.

Both black box testing and white box testing are important for ensuring the quality and reliability of a software system. Black box testing focuses on the external behavior of the software system, while white box testing focuses on the internal structure and design of the software system.

Black-box testing :

Black box testing is a testing approach that focuses on the external behavior of the software system without examining its internal workings. Testers who use this approach do not have knowledge of the internal code, design, or implementation of the software system being tested. Instead, they test the system based on its functional and non-functional requirements, and verify that it behaves as expected. This approach is often used to test the software from the user's perspective and to ensure that it meets the user's needs.

Functional testing: Testing the functionality of a software system without knowledge of the internal code or design. Example: testing a login page to ensure it accepts valid usernames and passwords, and rejects invalid ones.

Usability testing: Testing the user interface and user experience of a software system without knowledge of the internal code or design. Example: testing the ease of use and intuitiveness of a mobile app's navigation.

Performance testing: Testing the performance of a software system without knowledge of the internal code or design. Example: testing the load time and responsiveness of a web page under heavy traffic conditions.

Visual testing :

Visual testing is a technique used in software development to detect visual changes in the user interface of a software application. It involves comparing screenshots or images of the application at different points in time to identify any discrepancies.

Visual testing is particularly useful in applications that have a lot of graphical components or where the user interface is a critical aspect of the user experience. It can be used to detect issues such as misaligned elements, incorrect colors or fonts, broken layouts, and missing or incorrect images.

There are a number of tools available for performing visual testing, both manual and automated. Some popular automated tools include Applitools, Percy, and Visual Regression Testing. Manual visual testing can also be performed by simply comparing screenshots side-by-side.

Testing Levels

Unit testing :

Unit testing is a type of software testing where individual units or components of a software application are tested in isolation from the rest of the system. The purpose of unit testing is to validate that each unit of code performs as expected and to catch any errors early in the development process.

Unit tests typically involve writing code that tests a single function or method, with the test code often written by the developer who wrote the original function. The tests should be designed to cover all possible inputs and outputs for the function or method, and should verify that it behaves correctly under all conditions.

Unit tests are typically automated, and can be run as part of a continuous integration (CI) or continuous delivery (CD) pipeline to ensure that changes to the codebase don't introduce new bugs or regressions.

Some benefits of unit testing include:

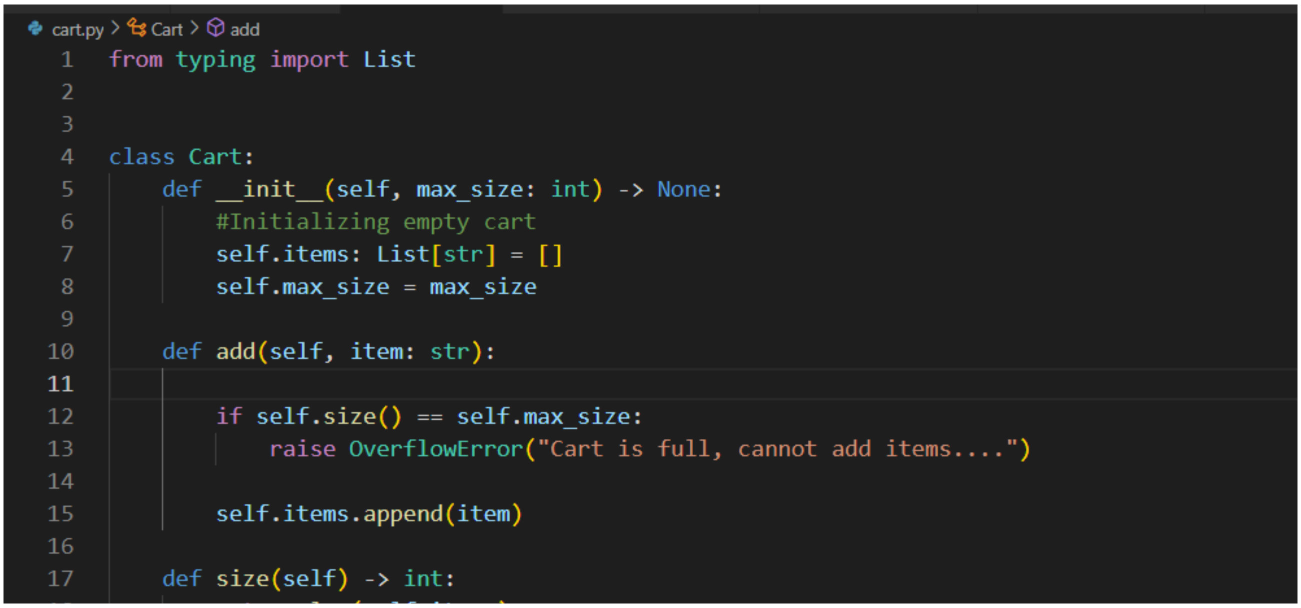
- Catching bugs early in the development process

- Ensuring that each component of the system works as intended

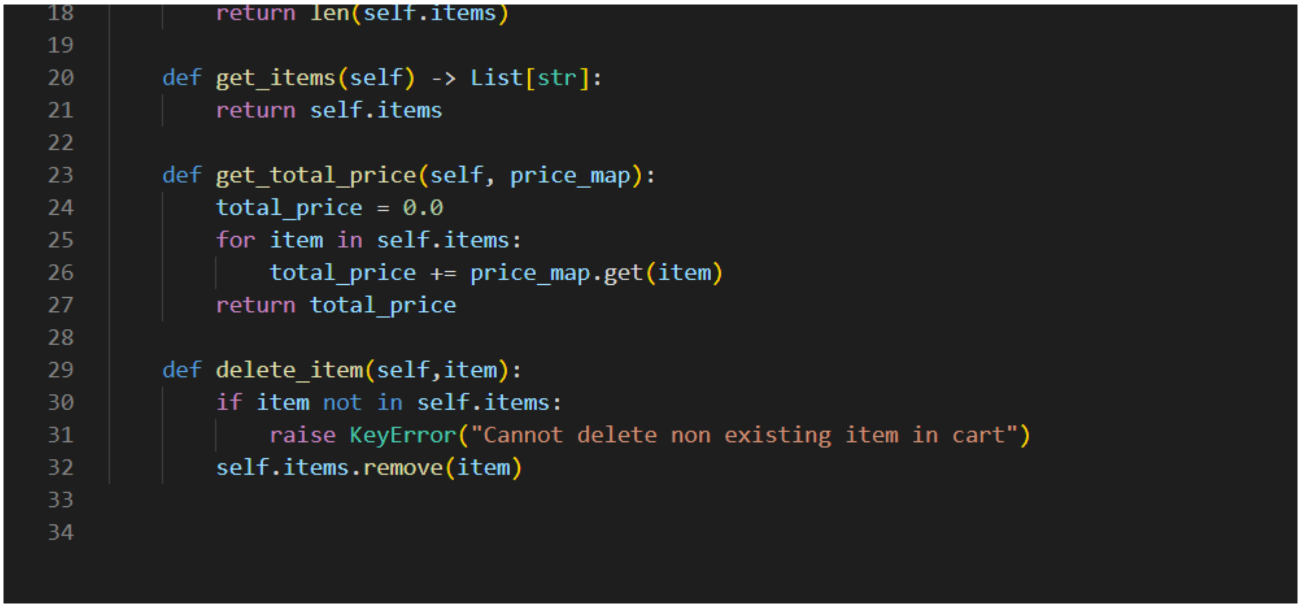
- Making it easier to refactor or modify code without breaking existing functionality

- Providing documentation and examples of how to use each function or method.

Unit testing is an important part of modern software development, and can help improve the quality and reliability of software applications.

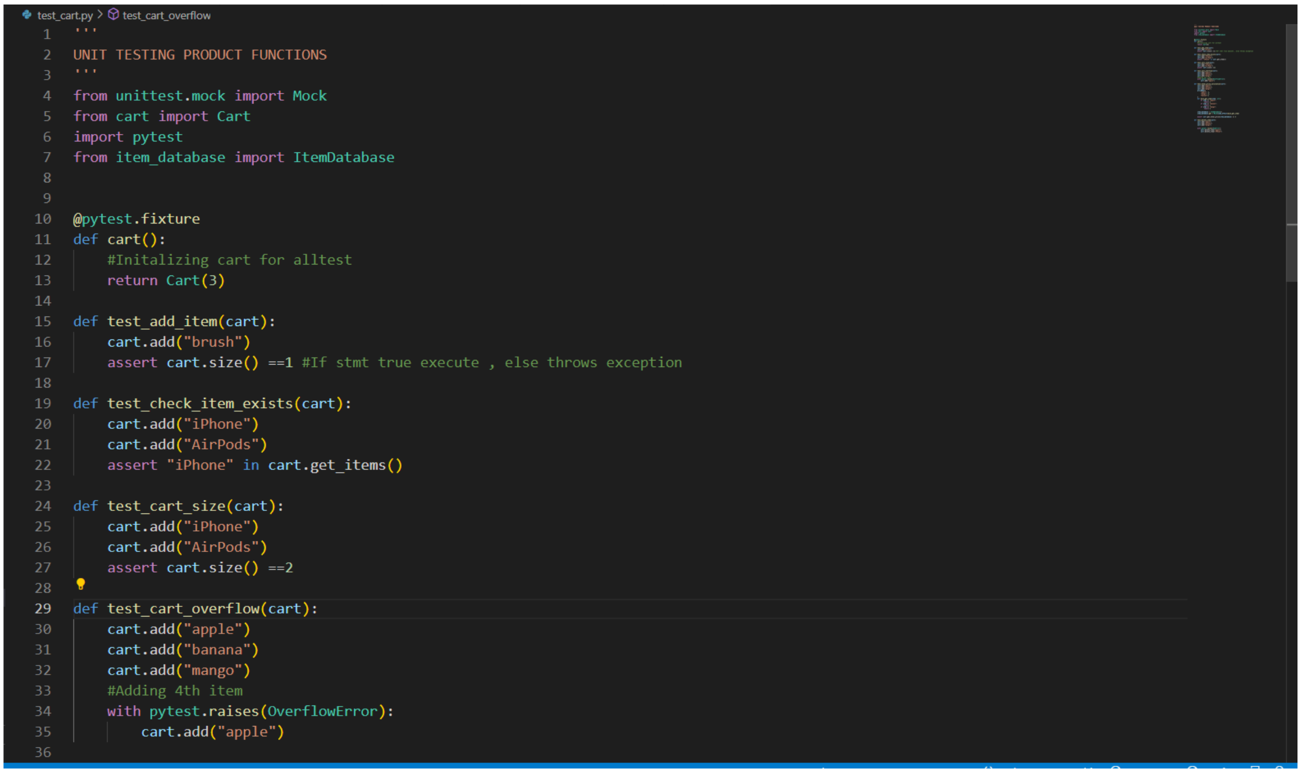


```python
from typing import List


class Cart:
    def __init__(self, max_size: int) -> None:
        #Initializing empty cart
        self.items: List[str] = []
        self.max_size = max_size

    def add(self, item: str):

        if self.size() == self.max_size:
            raise OverflowError("Cart is full, cannot add items....")

        self.items.append(item)

    def size(self) -> int:
```

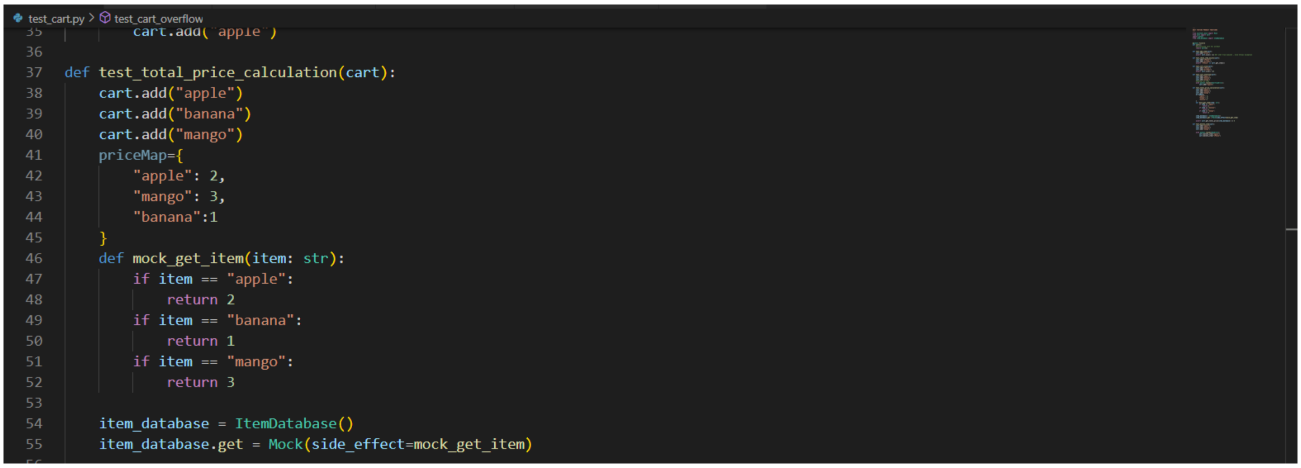


```python
        return len(self.items)

    def get_items(self) -> List[str]:
        return self.items

    def get_total_price(self, price_map):
        total_price = 0.0
        for item in self.items:
            total_price += price_map.get(item)
        return total_price

    def delete_item(self,item):
        if item not in self.items:
            raise KeyError("Cannot delete non existing item in cart")
        self.items.remove(item)


```

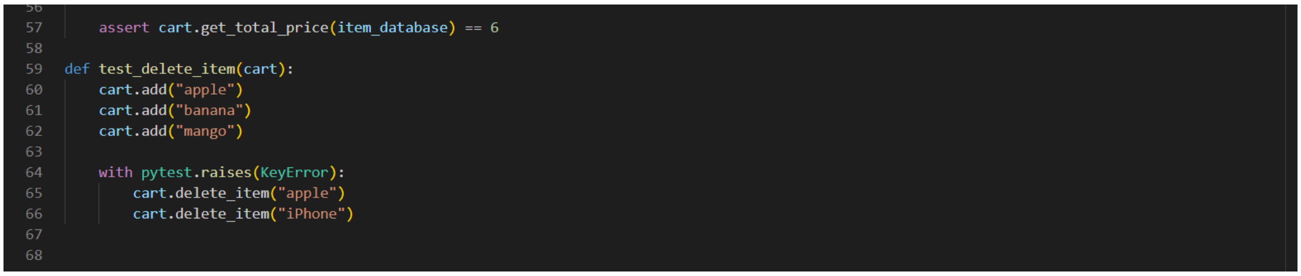
Test Script



```python
'''
UNIT TESTING PRODUCT FUNCTIONS
'''
from unittest.mock import Mock
from cart import Cart
import pytest
from item_database import ItemDatabase


@pytest.fixture
def cart():
    #Initalizing cart for alltest
    return Cart(3)

def test_add_item(cart):
    cart.add("brush")
    assert cart.size() ==1 #If stmt true execute , else throws exception

def test_check_item_exists(cart):
    cart.add("iPhone")
    cart.add("AirPods")
    assert "iPhone" in cart.get_items()

def test_cart_size(cart):
    cart.add("iPhone")
    cart.add("AirPods")
    assert cart.size() ==2

def test_cart_overflow(cart):
    cart.add("apple")
    cart.add("banana")
    cart.add("mango")
    #Adding 4th item
    with pytest.raises(OverflowError):
        cart.add("apple")

```

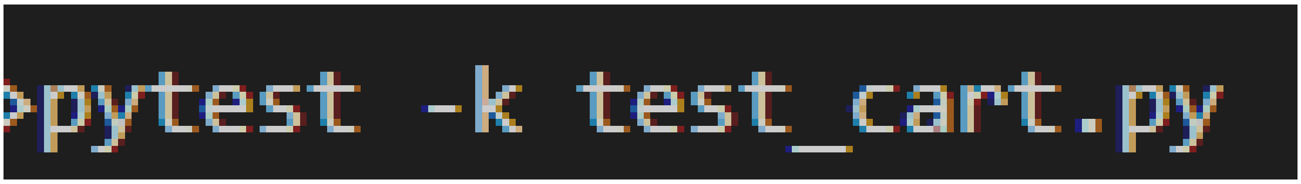


```python
        cart.add("apple")

def test_total_price_calculation(cart):
    cart.add("apple")
    cart.add("banana")
    cart.add("mango")
    priceMap={
        "apple": 2,
        "mango": 3,
        "banana":1
    }
    def mock_get_item(item: str):
        if item == "apple":
            return 2
        if item == "banana":
            return 1
        if item == "mango":
            return 3

    item_database = ItemDatabase()
    item_database.get = Mock(side_effect=mock_get_item)
```



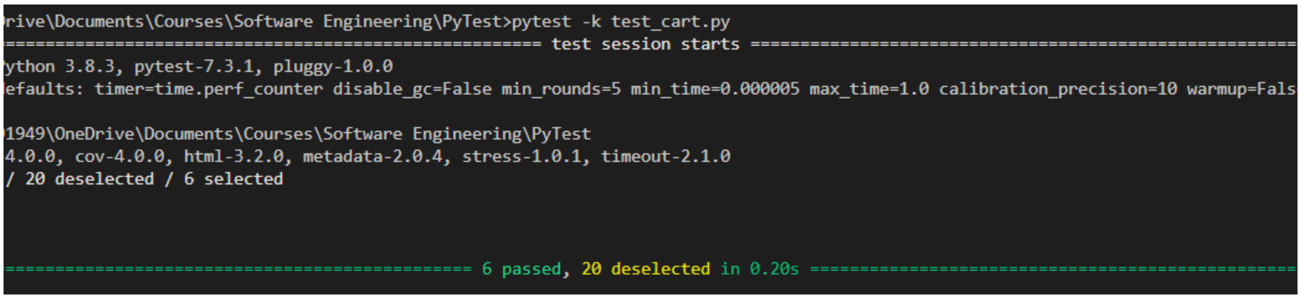
```
    assert cart.get_total_price(item_database) == 6

def test_delete_item(cart):
    cart.add("apple")
    cart.add("banana")
    cart.add("mango")

    with pytest.raises(KeyError):
        cart.delete_item("apple")
        cart.delete_item("iPhone")
```

Run Script



```
>pytest -k test_cart.py
```

Result



```
rive\Documents\Courses\Software Engineering\PyTest>pytest -k test_cart.py
====================================================== test session starts ======================================================
ython 3.8.3, pytest-7.3.1, pluggy-1.0.0
efaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=Fals

1949\OneDrive\Documents\Courses\Software Engineering\PyTest
4.0.0, cov-4.0.0, html-3.2.0, metadata-2.0.4, stress-1.0.1, timeout-2.1.0
/ 20 deselected / 6 selected



=============================================== 6 passed, 20 deselected in 0.20s ===============================================
```

Integration testing :

Integration testing is a type of software testing that focuses on verifying the correct interactions between different components or modules of a software system. The purpose of integration testing is to ensure that the individual modules or components work together as expected and that the system as a whole performs its intended functions.

Integration testing can be performed at different levels of granularity, depending on the complexity of the system and the level of detail required. Some common levels of integration testing include:

- Component integration testing: Testing the interaction between individual components or modules of the system.

- System integration testing: Testing the interaction between different subsystems or modules of the system.

- End-to-end integration testing: Testing the complete system, including all subsystems and external dependencies.

Integration testing is typically performed after unit testing and before system testing, as it verifies the interaction between different units of code. The testing may involve functional, performance, and security testing to ensure that the system behaves correctly under various scenarios and conditions.

Some benefits of integration testing include:

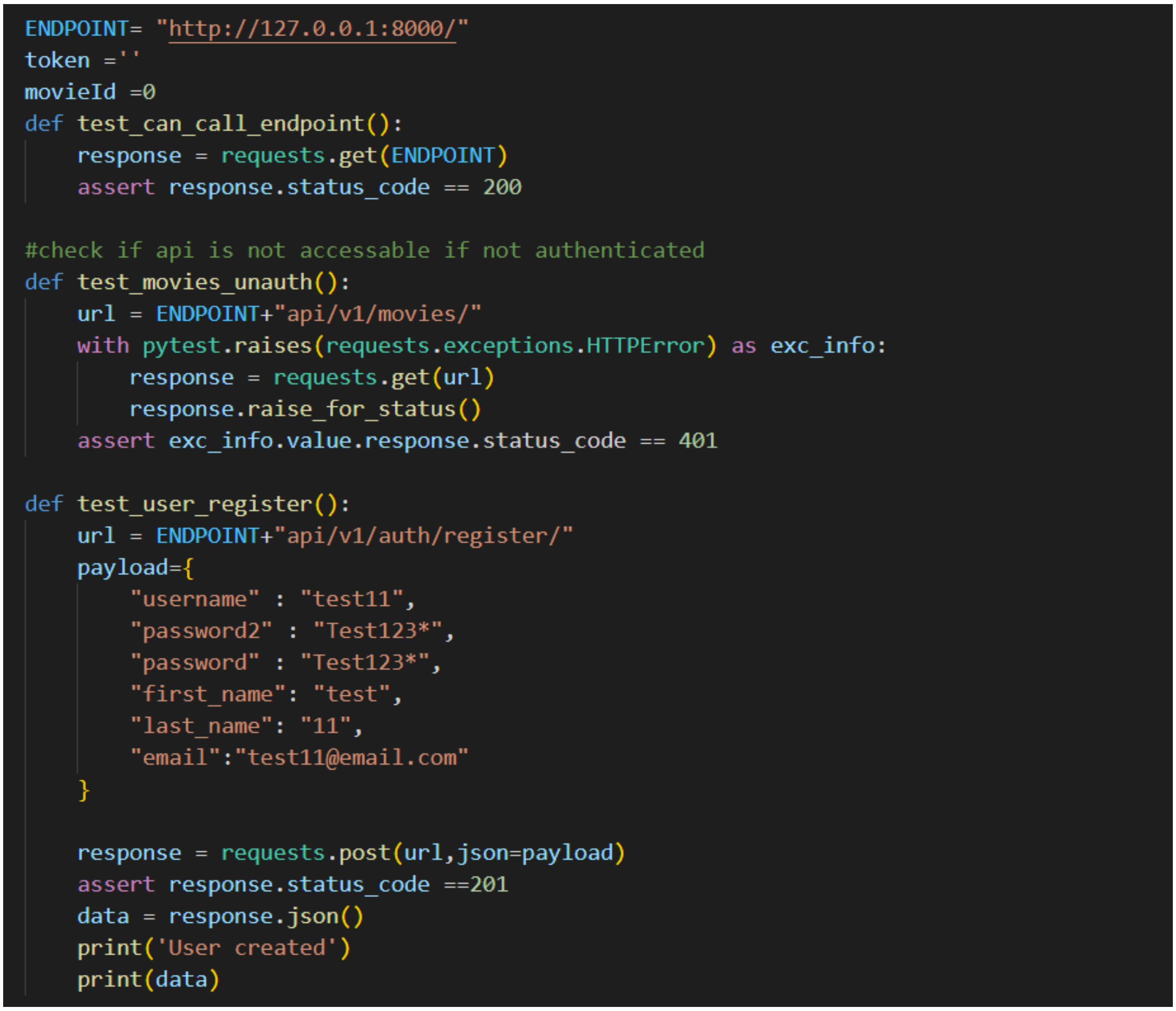
- Catching issues and defects early in the development process

- Ensuring that the system works as intended and meets the requirements

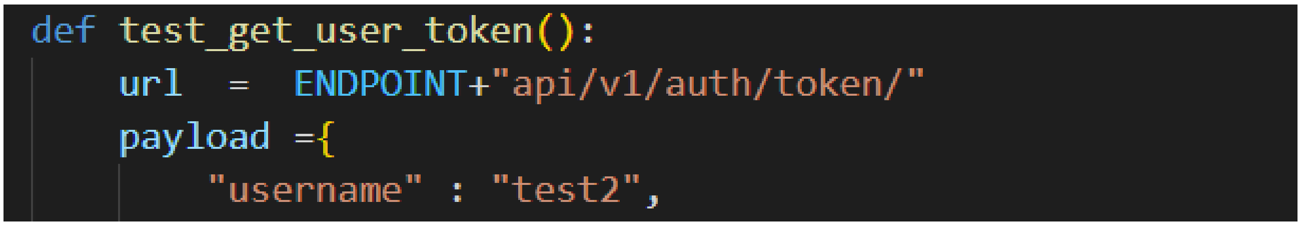
- Improving the reliability and quality of the system

- Reducing the cost and effort of testing by detecting issues early.

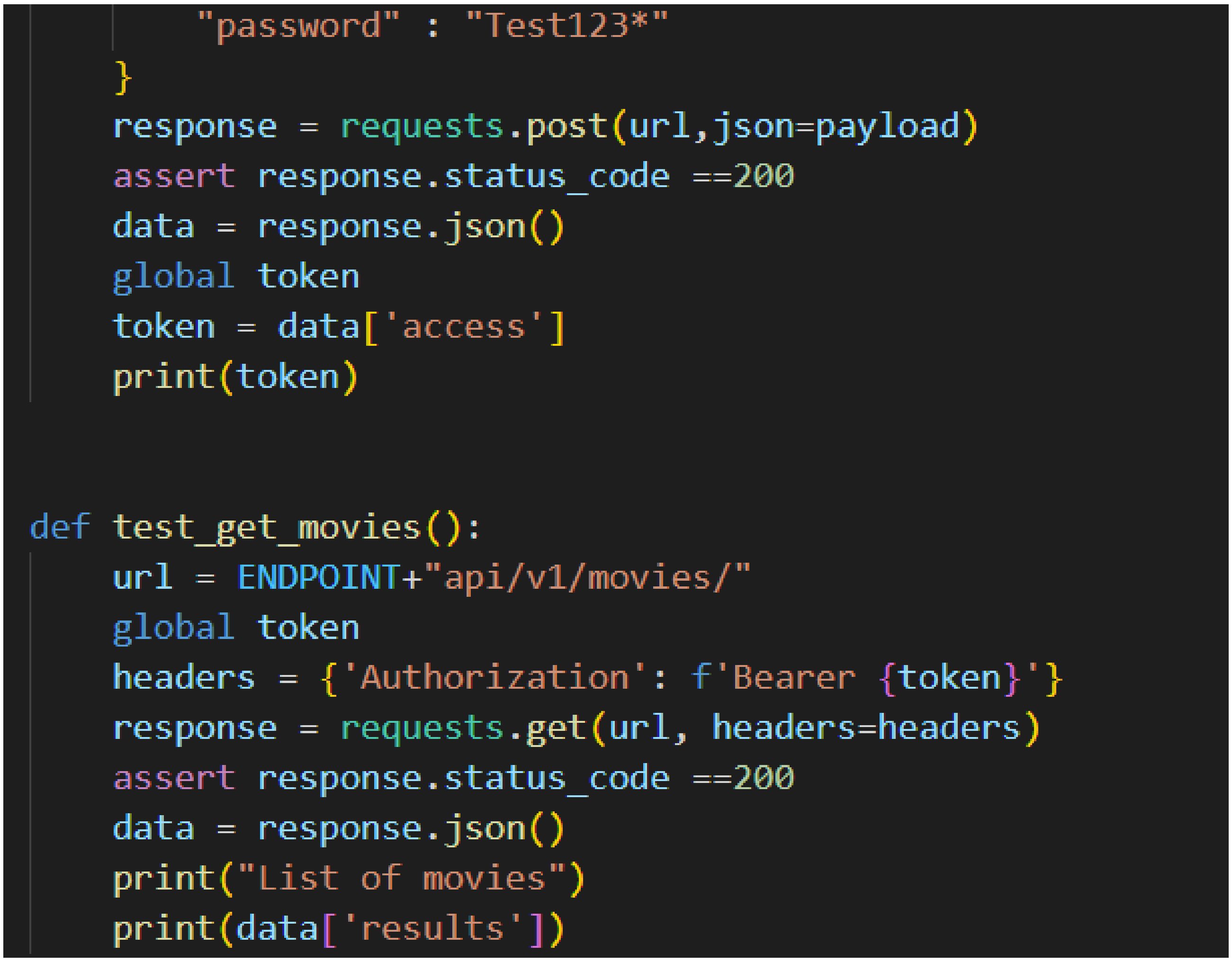
Integration testing is an essential part of the software development process, and is critical for ensuring that complex software systems work as intended.



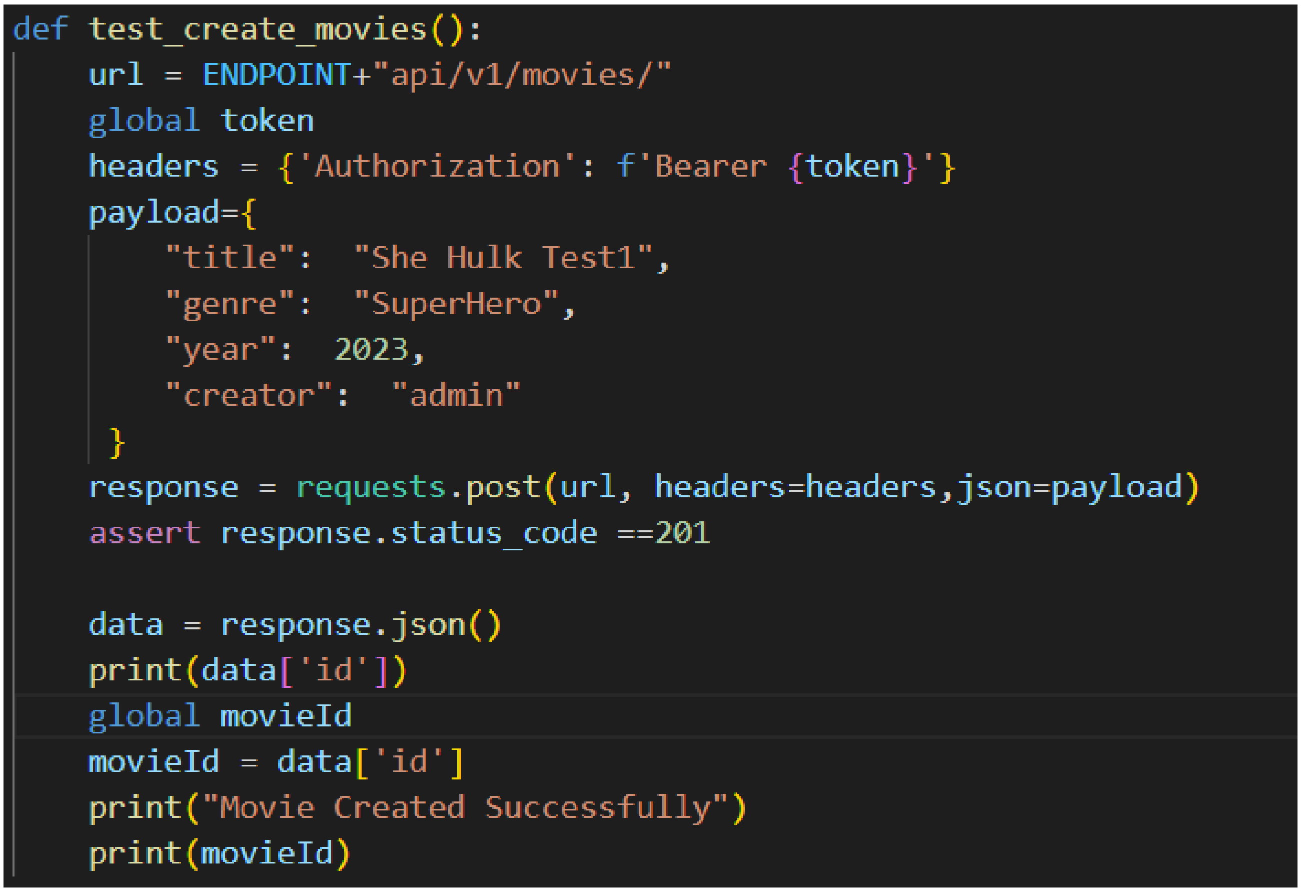
```
ENDPOINT= "http://127.0.0.1:8000/"
token =''
movieId =0
def test_can_call_endpoint():
    response = requests.get(ENDPOINT)
    assert response.status_code == 200

#check if api is not accessable if not authenticated
def test_movies_unauth():
    url = ENDPOINT+"api/v1/movies/"
    with pytest.raises(requests.exceptions.HTTPError) as exc_info:
        response = requests.get(url)
        response.raise_for_status()
    assert exc_info.value.response.status_code == 401

def test_user_register():
    url = ENDPOINT+"api/v1/auth/register/"
    payload={
        "username" : "test11",
        "password2" : "Test123*",
        "password" : "Test123*",
        "first_name": "test",
        "last_name": "11",
        "email":"test11@email.com"
    }

    response = requests.post(url,json=payload)
    assert response.status_code ==201
    data = response.json()
    print('User created')
    print(data)
```



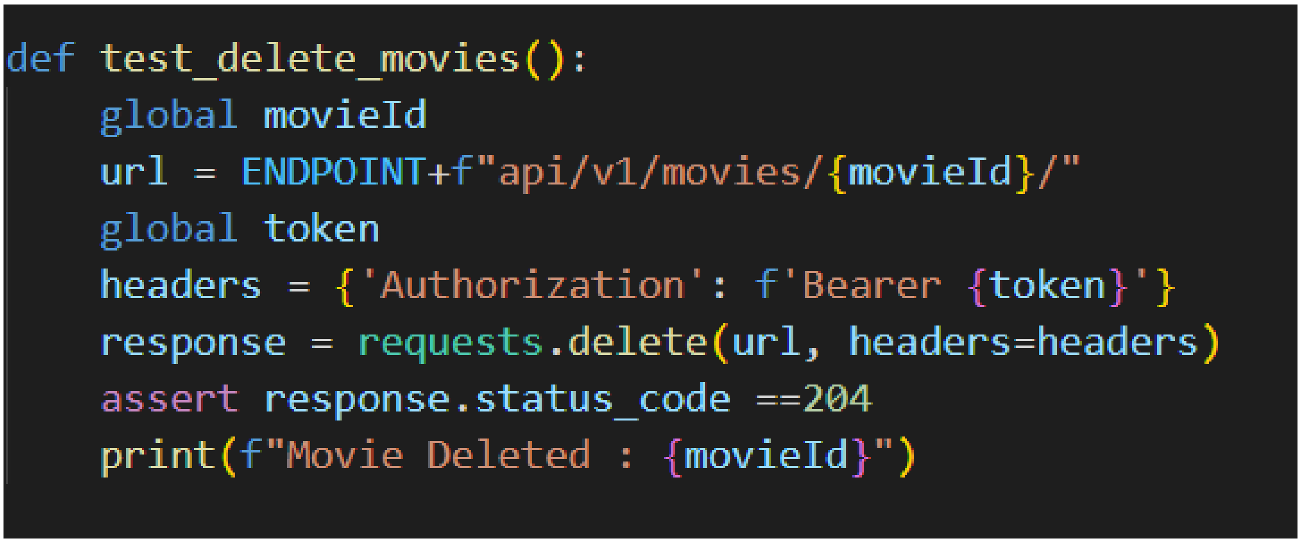
```
def test_get_user_token():
    url  =  ENDPOINT+"api/v1/auth/token/"
    payload ={
        "username" : "test2",
```



```
            "password" : "Test123*"
        }
        response = requests.post(url,json=payload)
        assert response.status_code ==200
        data = response.json()
        global token
        token = data['access']
        print(token)


def test_get_movies():
        url = ENDPOINT+"api/v1/movies/"
        global token
        headers = {'Authorization': f'Bearer {token}'}
        response = requests.get(url, headers=headers)
        assert response.status_code ==200
        data = response.json()
        print("List of movies")
        print(data['results'])
```

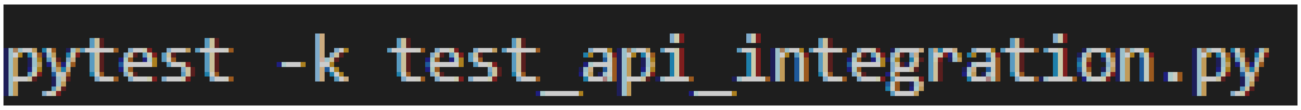


```
def test_create_movies():
    url = ENDPOINT+"api/v1/movies/"
    global token
    headers = {'Authorization': f'Bearer {token}'}
    payload={
        "title":  "She Hulk Test1",
        "genre":  "SuperHero",
        "year":  2023,
        "creator":  "admin"
    }
    response = requests.post(url, headers=headers,json=payload)
    assert response.status_code ==201

    data = response.json()
    print(data['id'])
    global movieId
    movieId = data['id']
    print("Movie Created Successfully")
    print(movieId)
```



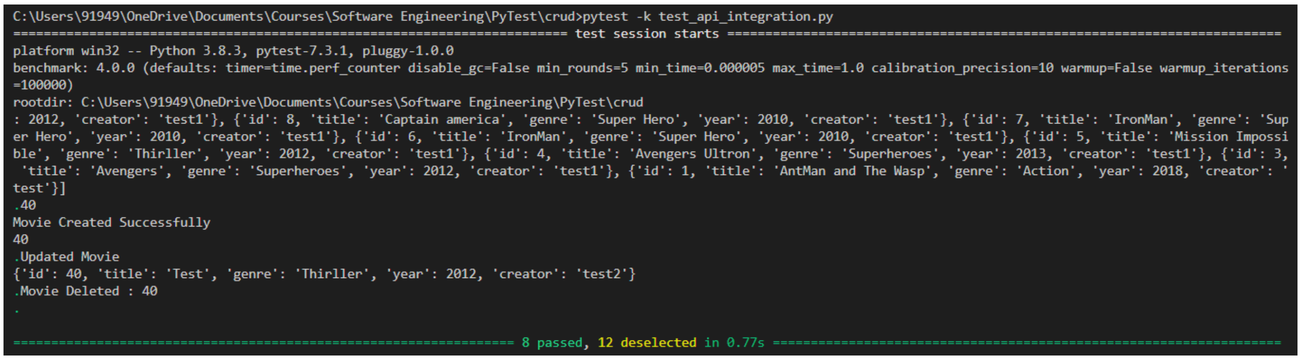
```python
def test_delete_movies():
    global movieId
    url = ENDPOINT+f"api/v1/movies/{movieId}/"
    global token
    headers = {'Authorization': f'Bearer {token}'}
    response = requests.delete(url, headers=headers)
    assert response.status_code ==204
    print(f"Movie Deleted : {movieId}")
```

Run Test



```
pytest -k test_api_integration.py
```

Result



```
C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud>pytest -k test_api_integration.py
=========================================================================== test session starts ===========================================================================
platform win32 -- Python 3.8.3, pytest-7.3.1, pluggy-1.0.0
benchmark: 4.0.0 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=False warmup_iterations
=100000)
rootdir: C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud
: 2012, 'creator': 'test1'}, {'id': 8, 'title': 'Captain america', 'genre': 'Super Hero', 'year': 2010, 'creator': 'test1'}, {'id': 7, 'title': 'IronMan', 'genre': 'Sup
er Hero', 'year': 2010, 'creator': 'test1'}, {'id': 6, 'title': 'IronMan', 'genre': 'Super Hero', 'year': 2010, 'creator': 'test1'}, {'id': 5, 'title': 'Mission Impossi
ble', 'genre': 'Thirller', 'year': 2012, 'creator': 'test1'}, {'id': 4, 'title': 'Avengers Ultron', 'genre': 'Superheroes', 'year': 2013, 'creator': 'test1'}, {'id': 3,
 'title': 'Avengers', 'genre': 'Superheroes', 'year': 2012, 'creator': 'test1'}, {'id': 1, 'title': 'AntMan and The Wasp', 'genre': 'Action', 'year': 2018, 'creator': '
test'}]
.40
Movie Created Successfully
40
.Updated Movie
{'id': 40, 'title': 'Test', 'genre': 'Thirller', 'year': 2012, 'creator': 'test2'}
.Movie Deleted : 40
.

================================================================= 8 passed, 12 deselected in 0.77s ==================================================================
```

System testing :

System testing is a type of software testing that evaluates the behavior of a complete software system or application as a whole, rather than testing its individual components in isolation. The goal of system testing is to ensure that the software system meets its functional and non-functional requirements and performs as expected in the real-world environment.

Here are some examples of system testing:

Integration testing: Testing the interaction and communication between the different components or modules of the software system.

Performance testing: Testing the speed, scalability, and stability of the software system under various workloads and usage scenarios.

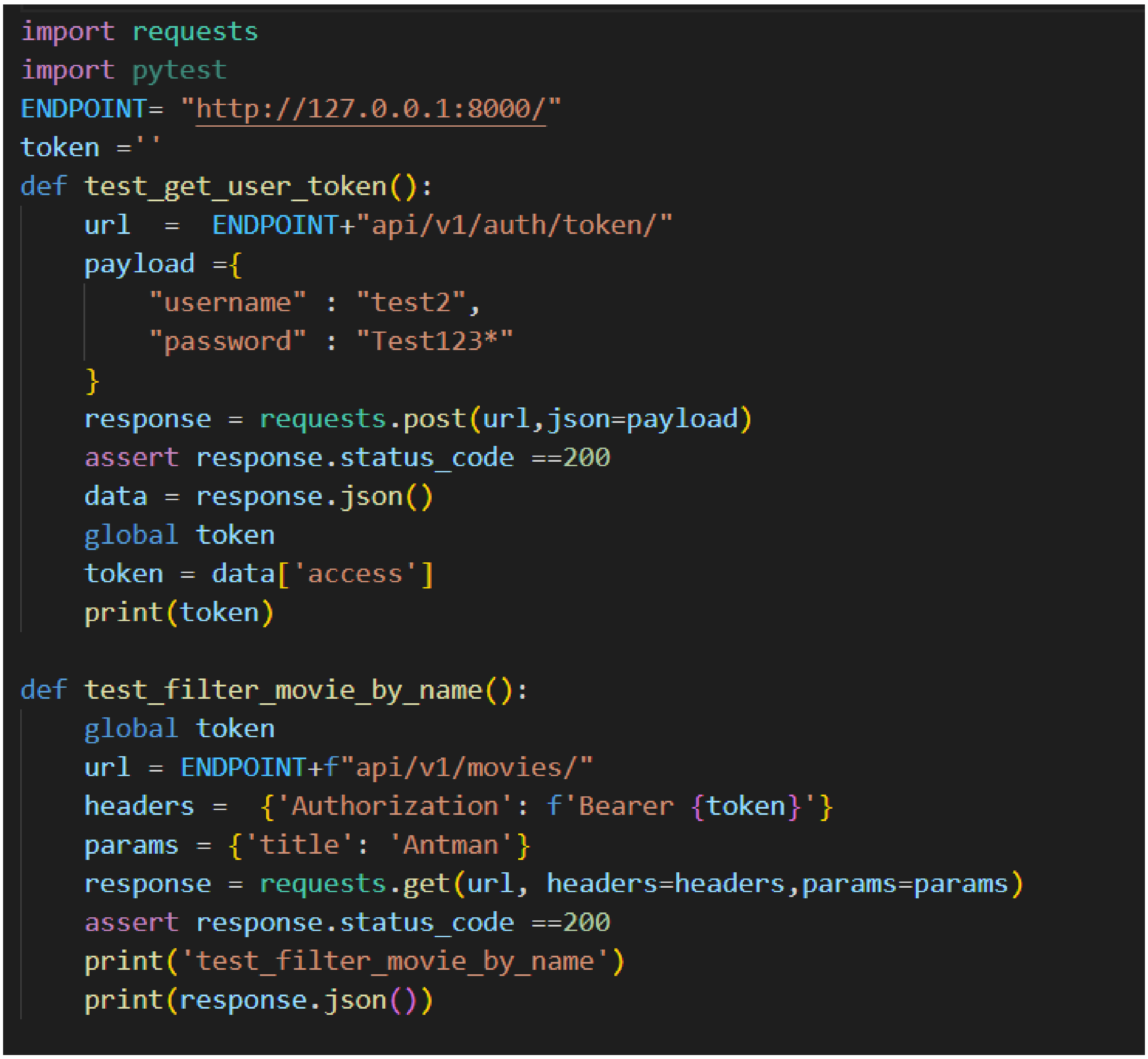
Security testing: Testing the software system for any vulnerabilities or weaknesses that could be exploited by attackers, and ensuring that it meets the relevant security standards and regulations.

User acceptance testing (UAT): Testing the software system with real end-users to ensure that it meets their needs and expectations, and provides a satisfactory user experience.

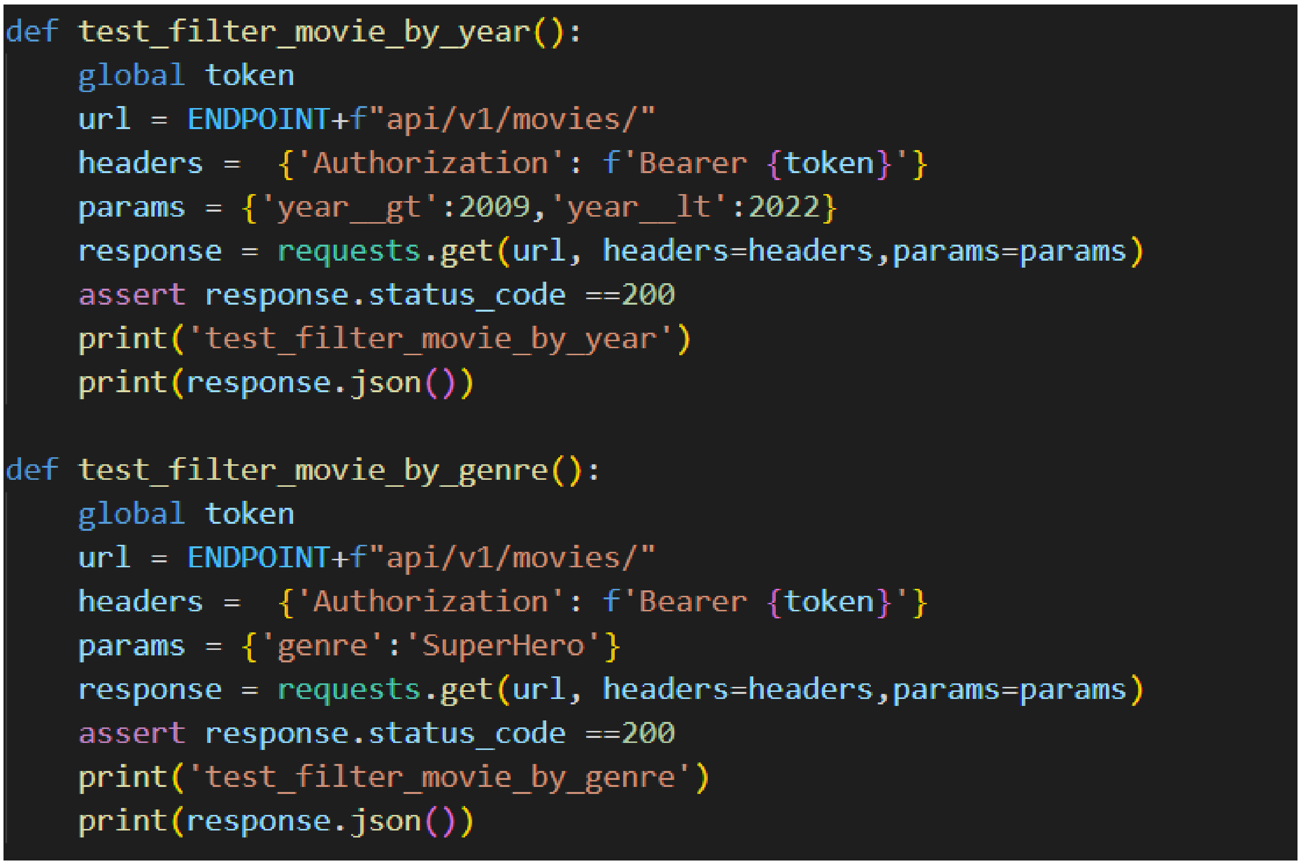
Compatibility testing: Testing the software system on different hardware, software, and network configurations to ensure that it works well in different environments.

Functional Testing: Functional testing is a sort of software testing, that assesses a software application's functionality, by comparing a software application's functionality to the functional requirements, specifications. Functional testing focuses on examining how the application's features, functionalities, and behavior will appear to end-users and that it meets the business and user requirements.

Test Script



```python
import requests
import pytest
ENDPOINT= "http://127.0.0.1:8000/"
token =''
def test_get_user_token():
    url  =  ENDPOINT+"api/v1/auth/token/"
    payload ={
        "username" : "test2",
        "password" : "Test123*"
    }
    response = requests.post(url,json=payload)
    assert response.status_code ==200
    data = response.json()
    global token
    token = data['access']
    print(token)

def test_filter_movie_by_name():
    global token
    url = ENDPOINT+f"api/v1/movies/"
    headers =  {'Authorization': f'Bearer {token}'}
    params = {'title': 'Antman'}
    response = requests.get(url, headers=headers,params=params)
    assert response.status_code ==200
    print('test_filter_movie_by_name')
    print(response.json())
```

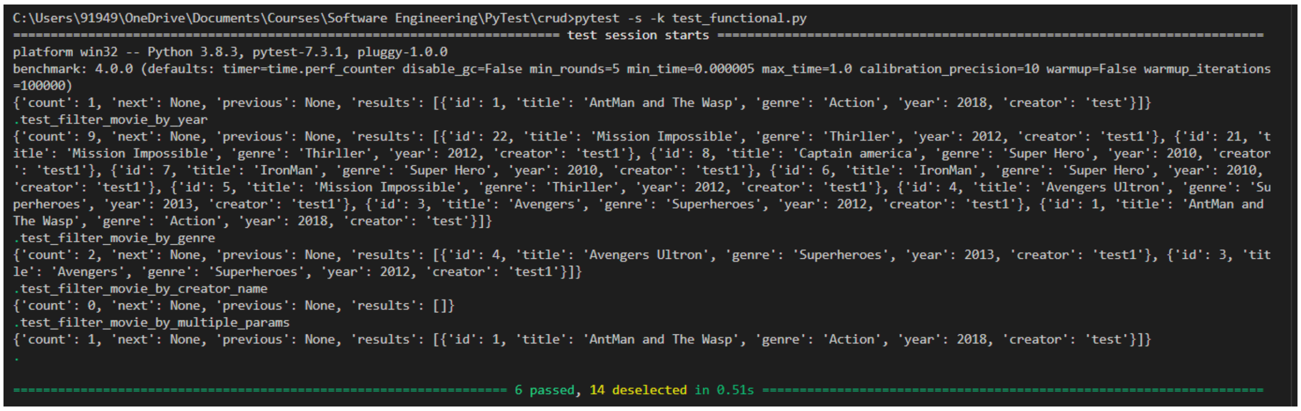


```python
def test_filter_movie_by_year():
    global token
    url = ENDPOINT+f"api/v1/movies/"
    headers =  {'Authorization': f'Bearer {token}'}
    params = {'year__gt':2009,'year__lt':2022}
    response = requests.get(url, headers=headers,params=params)
    assert response.status_code ==200
    print('test_filter_movie_by_year')
    print(response.json())

def test_filter_movie_by_genre():
    global token
    url = ENDPOINT+f"api/v1/movies/"
    headers =  {'Authorization': f'Bearer {token}'}
    params = {'genre':'SuperHero'}
    response = requests.get(url, headers=headers,params=params)
    assert response.status_code ==200
    print('test_filter_movie_by_genre')
    print(response.json())
```

Run Test

```
pytest -s -k test_functional.py
```

Results



```
C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud>pytest -s -k test_functional.py
========================================================================== test session starts ==========================================================================
platform win32 -- Python 3.8.3, pytest-7.3.1, pluggy-1.0.0
benchmark: 4.0.0 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=False warmup_iterations
=100000)
{'count': 1, 'next': None, 'previous': None, 'results': [{'id': 1, 'title': 'AntMan and The Wasp', 'genre': 'Action', 'year': 2018, 'creator': 'test'}]}
.test_filter_movie_by_year
{'count': 9, 'next': None, 'previous': None, 'results': [{'id': 22, 'title': 'Mission Impossible', 'genre': 'Thirller', 'year': 2012, 'creator': 'test1'}, {'id': 21, 't
itle': 'Mission Impossible', 'genre': 'Thirller', 'year': 2012, 'creator': 'test1'}, {'id': 8, 'title': 'Captain america', 'genre': 'Super Hero', 'year': 2010, 'creator
': 'test1'}, {'id': 7, 'title': 'IronMan', 'genre': 'Super Hero', 'year': 2010, 'creator': 'test1'}, {'id': 6, 'title': 'IronMan', 'genre': 'Super Hero', 'year': 2010,
'creator': 'test1'}, {'id': 5, 'title': 'Mission Impossible', 'genre': 'Thirller', 'year': 2012, 'creator': 'test1'}, {'id': 4, 'title': 'Avengers Ultron', 'genre': 'Su
perheroes', 'year': 2013, 'creator': 'test1'}, {'id': 3, 'title': 'Avengers', 'genre': 'Superheroes', 'year': 2012, 'creator': 'test1'}, {'id': 1, 'title': 'AntMan and
The Wasp', 'genre': 'Action', 'year': 2018, 'creator': 'test'}]}
.test_filter_movie_by_genre
{'count': 2, 'next': None, 'previous': None, 'results': [{'id': 4, 'title': 'Avengers Ultron', 'genre': 'Superheroes', 'year': 2013, 'creator': 'test1'}, {'id': 3, 'tit
le': 'Avengers', 'genre': 'Superheroes', 'year': 2012, 'creator': 'test1'}]}
.test_filter_movie_by_creator_name
{'count': 0, 'next': None, 'previous': None, 'results': []}
.test_filter_movie_by_multiple_params
{'count': 1, 'next': None, 'previous': None, 'results': [{'id': 1, 'title': 'AntMan and The Wasp', 'genre': 'Action', 'year': 2018, 'creator': 'test'}]}
.

=================================================================== 6 passed, 14 deselected in 0.51s ===================================================================
```

System testing is an important phase of the software development lifecycle and helps to ensure that the software system meets its intended requirements and performs as expected in the real-world environment.

Operational Acceptance testing :

Operational acceptance testing (OAT) is a type of testing that focuses on ensuring that a software system is able to function correctly and effectively in its intended operational environment. When it comes to testing APIs, Pytest can be used to perform OAT by simulating real-world scenarios and ensuring that the API is able to handle expected traffic and usage patterns.

Here are some steps for performing OAT using Pytest for APIs:

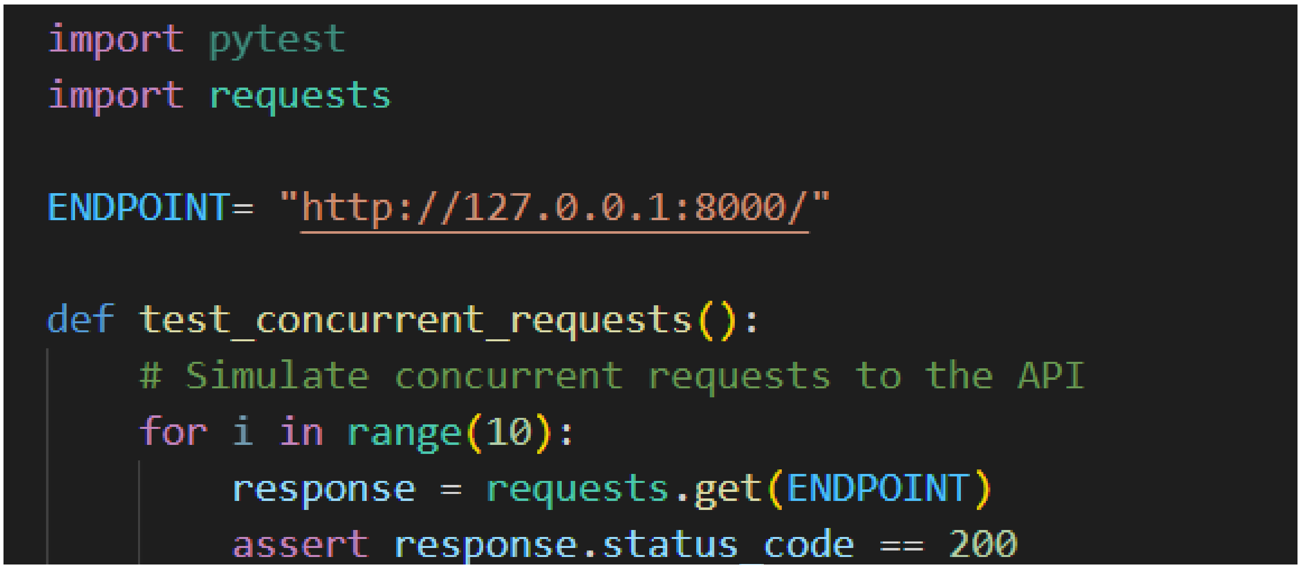
Define the operational environment: Identify the key factors that will impact the performance and behavior of the API in its intended operational environment. This may include factors such as network latency, user traffic, and server load.

Develop test scenarios: Develop test scenarios that simulate real-world usage patterns and traffic for the API. This may include scenarios such as concurrent user requests, long-running transactions, and error handling.

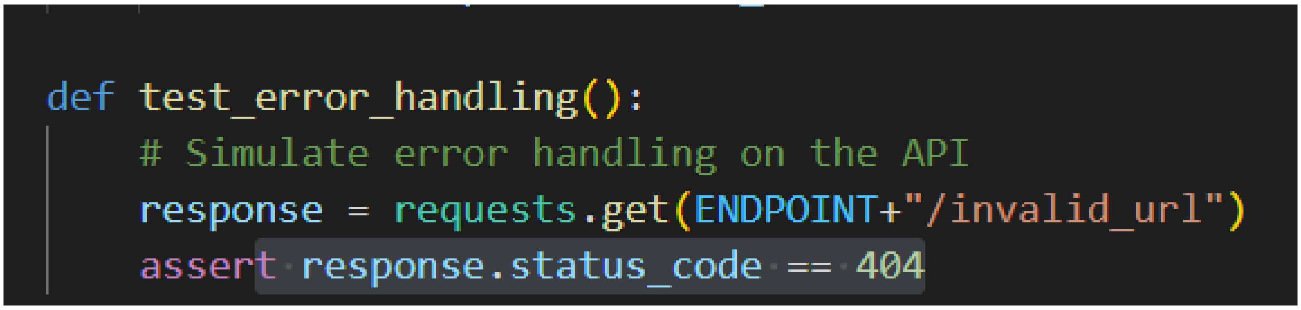
Implement tests using Pytest: Write tests using Pytest that simulate the test scenarios identified in step 2. These tests should be designed to stress-test the API and ensure that it is able to handle expected usage patterns and traffic.

Run tests and analyze results: Run the tests and analyze the results to identify any areas where the API may be experiencing performance issues or other problems. Use this information to identify areas for improvement and optimization.

By using Pytest to perform OAT for APIs, developers and testers can ensure that the API is able to perform effectively and reliably in its intended operational environment. This can help to minimize the risk of downtime, errors, and other issues that can impact the user experience and damage the reputation of the software system. Operational Acceptance Testing is an important part of the software development process, and is critical for ensuring that a software system is ready for deployment and can operate effectively in its intended production environment.

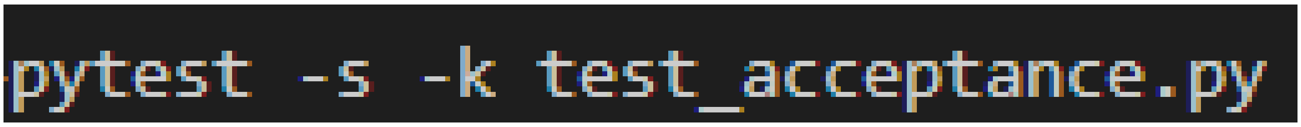


```python
import pytest
import requests

ENDPOINT= "http://127.0.0.1:8000/"

def test_concurrent_requests():
    # Simulate concurrent requests to the API
    for i in range(10):
        response = requests.get(ENDPOINT)
        assert response.status_code == 200
```



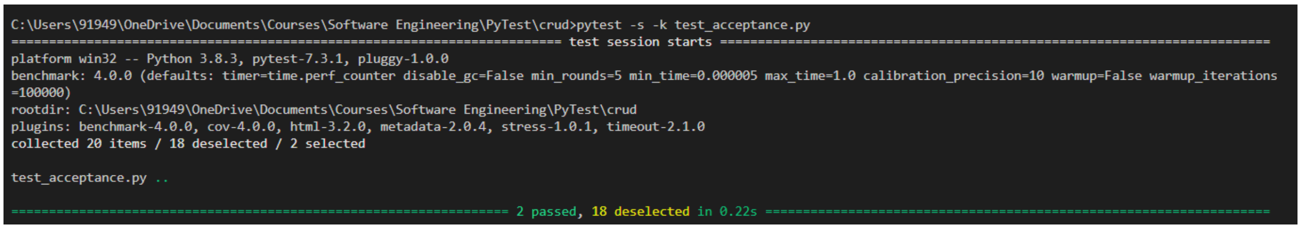
```python
def test_error_handling():
    # Simulate error handling on the API
    response = requests.get(ENDPOINT+"/invalid_url")
    assert response.status_code == 404
```

Run script



```
pytest -s -k test_acceptance.py
```

Result



```
C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud>pytest -s -k test_acceptance.py
========================================================================= test session starts ==========================================================================
platform win32 -- Python 3.8.3, pytest-7.3.1, pluggy-1.0.0
benchmark: 4.0.0 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=False warmup_iterations
=100000)
rootdir: C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud
plugins: benchmark-4.0.0, cov-4.0.0, html-3.2.0, metadata-2.0.4, stress-1.0.1, timeout-2.1.0
collected 20 items / 18 deselected / 2 selected

test_acceptance.py ..

================================================================== 2 passed, 18 deselected in 0.22s ===================================================================
```

Testing Types

Installation and compatibility testing

Installation and compatibility testing are crucial aspects of software testing that ensure that software can be successfully installed and run on different platforms and configurations. Without proper installation and compatibility testing, software may not function as intended, leading to user frustration, wasted time, and resources.

Pytest is a powerful and flexible testing framework that can be used for installation and compatibility testing. It offers a wide range of features and plugins that make it easy to manage complex test environments and configurations, as well as to generate detailed reports of test results.

Regression Testing

Regression testing is a type of software testing that is used to verify that changes made to an application or system do not have any unintended consequences or side effects on existing functionality. It involves retesting existing features and functionality of the system to ensure that they are still working as expected after a new release or change has been made.

In regression testing, test cases that were previously developed to test the application are re-executed to ensure that the new changes have not introduced any new defects or caused any existing functionality to fail. This helps ensure that the application still meets the requirements and specifications it was originally designed for.

Pytest can be used for regression testing by creating test cases that cover the functionality of the application and then running them against different versions of the codebase. Pytest makes it easy to create and organize test cases, and its reporting features make it simple to identify any issues or regressions that may arise.

By using Pytest for regression testing, developers can ensure that changes made to the application are thoroughly tested and validated before being released to users, helping to prevent any unintended consequences or negative impact on the user experience.

Continuous And Destructive Testing

Pytest is a flexible and powerful testing framework for Python that can be used for both continuous and destructive testing.

Continuous testing: It is a testing approach in which tests are run automatically and continuously as code changes are made. The goal of continuous testing is to catch errors and bugs early in the development process, before they can cause more serious problems. Pytest can be integrated into continuous integration (CI) tools like Jenkins, Travis CI, or CircleCI to run tests automatically every time code changes are pushed to a repository.

Destructive testing: on the other hand, is a type of testing in which the system is intentionally pushed to its limits or beyond, in order to test its robustness and resilience. Destructive testing can help uncover hidden bugs and vulnerabilities that might not be apparent in normal usage scenarios. Pytest can be used to write destructive tests that intentionally stress the system and its components.

Here are some examples of how Pytest can be used for continuous and destructive testing:

Continuous testing: Pytest can be integrated into a CI/CD pipeline to automatically run tests every time code changes are made. This can help catch errors and bugs early in the development process, and ensure that changes don't break existing functionality.

Destructive testing: Pytest can be used to write tests that intentionally stress the system, such as by sending large amounts of data, making multiple simultaneous requests, or simulating network failures. These tests can help uncover hidden bugs and vulnerabilities that might not be apparent in normal usage scenarios.

Overall, Pytest is a flexible and powerful testing framework that can be used for both continuous and destructive testing, depending on your testing needs. By leveraging Pytest's features and integrations, you can ensure that your code is well-tested and reliable, no matter what challenges it might face.

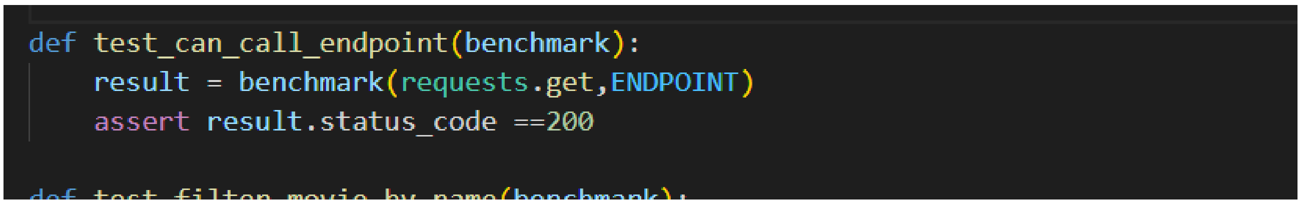
System performance and Usability Testing

System performance testing and usability testing are two different types of testing that can be performed using Pytest.

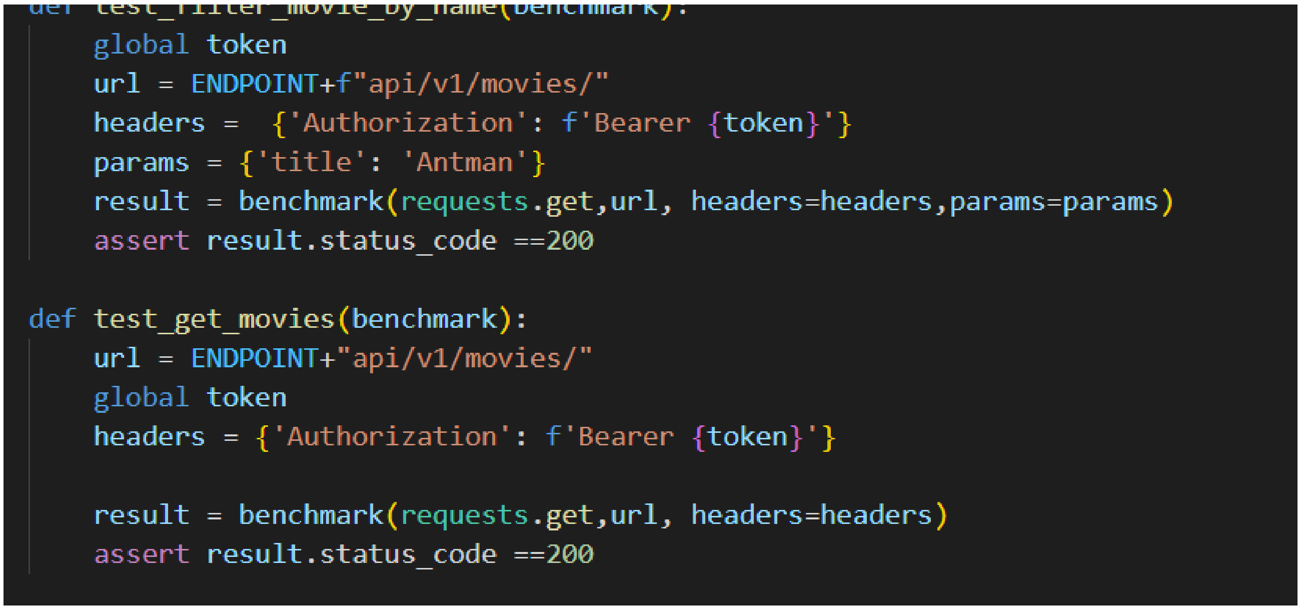
System performance testing involves testing the performance and scalability of the system under varying load conditions, to ensure that the system can handle the expected load and perform well under stress. Pytest can be used for system performance testing by leveraging plugins like pytest-benchmark, which allows you to measure and compare the performance of different code paths and functions.

Usability testing, on the other hand, involves testing the usability and user-friendliness of the system, to ensure that the system is easy to use and meets the needs of its users. Pytest can be used for usability testing by leveraging plugins like pytest-selenium, which allows you to write automated tests for web applications using Selenium WebDriver, and pytest-qt, which allows you to write automated tests for GUI applications using the PyQt framework.

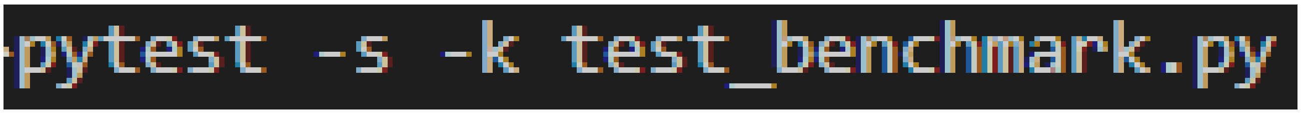
By using Pytest for system performance and usability testing, you can ensure that your system is both performant and user-friendly, helping to improve the overall quality and user experience of the system.



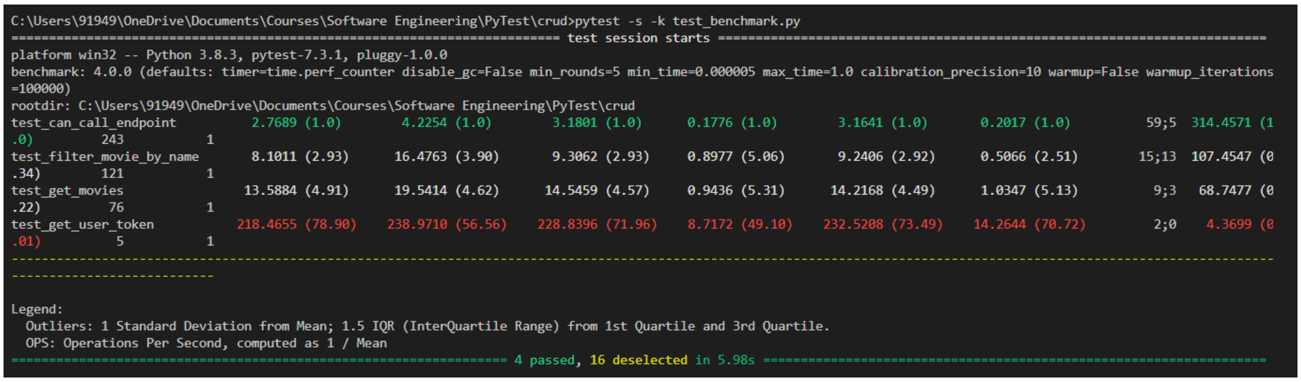
```python
def test_can_call_endpoint(benchmark):
    result = benchmark(requests.get,ENDPOINT)
    assert result.status_code ==200

def test_filter_movie_by_name(benchmark):
```



```
def test_filter_movie_by_name(benchmark):
    global token
    url = ENDPOINT+f"api/v1/movies/"
    headers =  {'Authorization': f'Bearer {token}'}
    params = {'title': 'Antman'}
    result = benchmark(requests.get,url, headers=headers,params=params)
    assert result.status_code ==200

def test_get_movies(benchmark):
    url = ENDPOINT+"api/v1/movies/"
    global token
    headers = {'Authorization': f'Bearer {token}'}

    result = benchmark(requests.get,url, headers=headers)
    assert result.status_code ==200
```



```
pytest -s -k test_benchmark.py
```



```
C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud>pytest -s -k test_benchmark.py
======================================================================= test session starts ========================================================================
platform win32 -- Python 3.8.3, pytest-7.3.1, pluggy-1.0.0
benchmark: 4.0.0 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=False warmup_iterations
=100000)
rootdir: C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud
test_can_call_endpoint         2.7689 (1.0)        4.2254 (1.0)        3.1801 (1.0)       0.1776 (1.0)        3.1641 (1.0)       0.2017 (1.0)          59;5  314.4571 (1
.0)          243           1
test_filter_movie_by_name      8.1011 (2.93)      16.4763 (3.90)       9.3062 (2.93)      0.8977 (5.06)       9.2406 (2.92)      0.5066 (2.51)        15;13  107.4547 (0
.34)         121           1
test_get_movies               13.5884 (4.91)      19.5414 (4.62)      14.5459 (4.57)      0.9436 (5.31)      14.2168 (4.49)      1.0347 (5.13)          9;3   68.7477 (0
.22)          76           1
test_get_user_token          218.4655 (78.90)    238.9710 (56.56)    228.8396 (71.96)     8.7172 (49.10)    232.5208 (73.49)    14.2644 (70.72)         2;0    4.3699 (0
.01)           5           1
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------

Legend:
  Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quartile.
  OPS: Operations Per Second, computed as 1 / Mean
================================================================= 4 passed, 16 deselected in 5.98s =================================================================
```

Security Testing:

Security testing is a crucial aspect of software testing that ensures that the software is secure and can withstand potential security threats. It involves a wide range of testing techniques and methodologies, including vulnerability scanning, penetration testing, and security code review.

Security testing for APIs involves testing the security of the API endpoints, data validation, input validation, authentication and authorization mechanisms, and potential vulnerabilities like SQL injection, cross-site scripting (XSS), and other web application vulnerabilities.

In our testing project the movies APIs are accessible only when user is authenticated. Here we tested different scenarios for the APIs before and after authentication. Which enables us to understand whether the authentication and authorization is strictly imposed on APIs.

Development Testing:

Development testing is a type of software testing that is performed during the development process to identify and fix defects early on in the development lifecycle. This type of testing is critical for ensuring that software is of high quality and meets the requirements of its stakeholders.

Pytest is a powerful testing framework that can be used for development testing. With Pytest, developers can write automated tests for their code to ensure that it is working as expected and to catch defects before they make it to production. Pytest provides a simple, yet powerful syntax for writing tests and supports a wide variety of testing scenarios, including unit testing, integration testing, and functional testing. Pytest also provides a range of plugins and extensions that can be used to enhance the testing capabilities of the framework. For example, the pytest-mock plugin can be used for mocking external dependencies and systems during testing, while the pytest-cov plugin can be used for measuring test coverage.

By using Pytest for development testing, developers can ensure that their code is of high quality, meets the requirements of stakeholders, and is free of defects before it is released to production. This can help to save time and resources by catching defects early in the development process, and improve the overall quality and reliability of the software.

Testing Plugins

pytest-html :

Pytest-HTML is a plugin for Pytest that generates an HTML report of test results. It is a popular plugin that allows developers to view their test results in a browser, making it easier to understand and analyze test results.

When Pytest-HTML is installed, it generates an HTML report of the test results after each test run. The report includes information such as the test name, status (pass or fail), duration, and any error messages. The report can be viewed in a web browser, allowing developers to navigate and filter the results.

Pytest-HTML also provides several configuration options for customizing the report's appearance and content. Developers can configure the plugin to include or exclude specific information, such as captured output or screenshots, and to use custom templates for styling the report.

To use Pytest-HTML, developers need to install the plugin via pip or another package manager. Once installed, the plugin can be enabled by passing the `--html` flag when running Pytest. The plugin will automatically generate an HTML report of the test results and save it to a file.

In summary, Pytest-HTML is a plugin for Pytest that generates an HTML report of test results. It provides a user-friendly interface for viewing and analyzing test results and includes several configuration options for customizing the report's appearance and content.

pytest-benchmarks :

Pytest-benchmark is a plugin for Pytest that allows developers to benchmark the performance of their code. It is used to measure how long it takes for a specific function or block of code to execute and can help identify performance issues in code.

When Pytest-benchmark is installed, it provides a simple way to measure the time it takes to run a specific function or block of code. Developers can use Pytest-benchmark to run a function multiple times and measure the average time it takes to execute. This provides more accurate performance data than measuring the time it takes to run a function only once.

Pytest-benchmark also provides several configuration options for customizing the benchmarking process. Developers can set the number of runs, the time between runs, and the number of warm-up runs to perform. They can also specify the output format of the benchmark results.

To use Pytest-benchmark, developers need to install the plugin via pip or another package manager. Once installed, they can decorate the function or block of code they want to benchmark with the `@pytest_benchmark.fixture` decorator. Pytest-benchmark will automatically run the benchmark and generate a report of the results.

In summary, Pytest-benchmark is a plugin for Pytest that allows developers to benchmark the performance of their code. It provides a simple way to measure the time it takes to execute a specific function or block of code and includes several configuration options for customizing the benchmarking process.

pytest-cov :

Pytest-cov is a plugin for Pytest that provides coverage analysis for Python code. It is used to measure how much of the codebase is covered by tests and can help identify areas of the code that need additional testing.

When Pytest-cov is installed, it provides a simple way to measure code coverage for a specific test run. The plugin generates a report that shows which lines of code were executed during the test run and which lines were not executed. This provides a clear picture of which parts of the code are covered by tests and which parts are not.

Pytest-cov also provides several configuration options for customizing the coverage analysis. Developers can specify which files or directories to include or exclude from the analysis and can set thresholds for coverage percentages. They can also specify the output format of the coverage report.

To use Pytest-cov, developers need to install the plugin via pip or another package manager. Once installed, they can enable the plugin by passing the `--cov` flag when running Pytest. The plugin will automatically generate a coverage report for the test run.

In summary, Pytest-cov is a plugin for Pytest that provides coverage analysis for Python code. It measures code coverage for a specific test run and generates a report that shows which lines of code were executed during the test run and which lines were not executed. Pytest-cov also includes several configuration options for customizing the coverage analysis.

pytest-stress:

pytest-stress is a plugin for Pytest that provides tools for stress testing applications. It allows you to run Pytest tests repeatedly with varying inputs to simulate high loads on an application and check its stability and reliability.

The plugin provides a @pytest.mark.stress decorator that can be used to mark specific test functions for stress testing. This decorator accepts several options, including max_runs and timeout, which allow you to control how many times the test is run and how long it should be allowed to run before timing out.

Inside the test function, you can use the StressTest class provided by pytest-stress to generate random inputs for your application, and then assert that the output is correct.

StressTest provides several methods for generating random inputs, including integers, floats, strings, and more.

Using pytest-stress, you can simulate a high load on your application and ensure that it remains stable and reliable even under heavy usage. You can also use pytest-stress to identify performance bottlenecks or other issues that might only become apparent under high loads. Overall, pytest-stress is a powerful tool for testing the robustness and scalability of your applications.

We can run Test scripts in loop by using the following command **pytest --minutes 45.** This will run the scripts in loop for 45 minutes.

pytest-timeout:

pytest-timeout is a Pytest plugin that allows you to set timeouts for individual tests. It provides a simple and flexible way to specify a maximum time that a test can run, and if the test takes longer than that time, it will be aborted and marked as a failure.

The plugin adds a new @pytest.mark.timeout marker that can be used to set a timeout for a test. You can specify the timeout duration in seconds, and also provide a custom error message to be displayed if the test times out.

pytest-timeout is particularly useful when testing applications that involve long-running or potentially infinite processes, such as network communication, database queries, or complex algorithms. By setting a timeout for these tests, you can ensure that they complete within a reasonable time frame, and avoid blocking other tests from running.

In addition, pytest-timeout can be combined with other Pytest plugins, such as pytest-xdist and pytest-parallel, to run tests in parallel and scale your test suite to handle larger workloads.

Overall, pytest-timeout is a powerful tool for ensuring that your tests complete in a timely manner, and can help you identify and fix issues with slow or inefficient code.

```
pytest --timeout=300

@pytest.mark.timeout(60)

def test_foo():

    pass
```

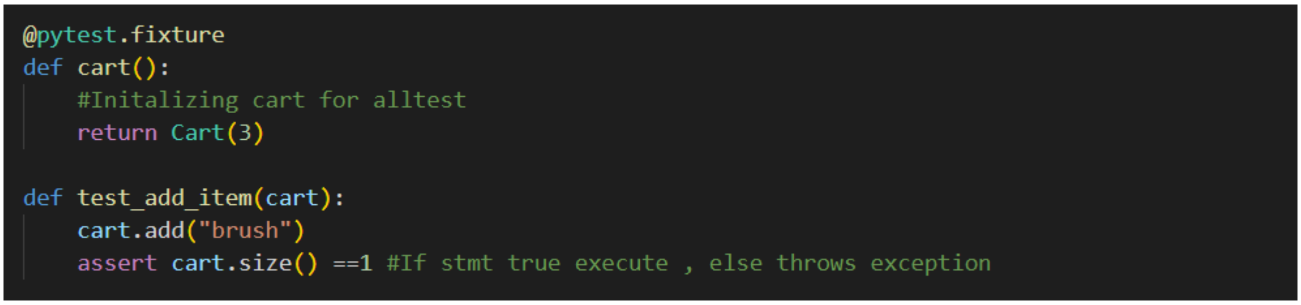
Testing Overall

Powerful features in Pytest:

fixtures :

Fixtures are a way to provide test data and objects to test functions. They are functions that are called before the test function and can return any object that the test function needs. Fixtures can be used to set up database connections, create test files, or generate mock objects, among other things.

For example, if a test function requires a database connection, a fixture can be created that creates the connection and returns it to the test function. This ensures that the database connection is available for the test and is closed after the test is finished.



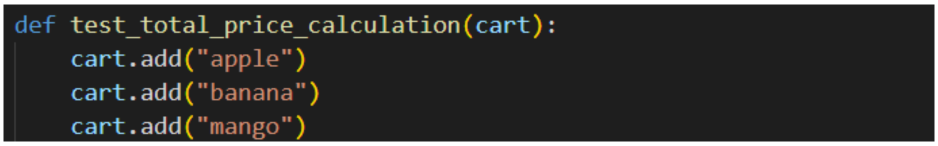
```
@pytest.fixture
def cart():
    #Initalizing cart for alltest
    return Cart(3)

def test_add_item(cart):
    cart.add("brush")
    assert cart.size() ==1 #If stmt true execute , else throws exception
```

Mock :

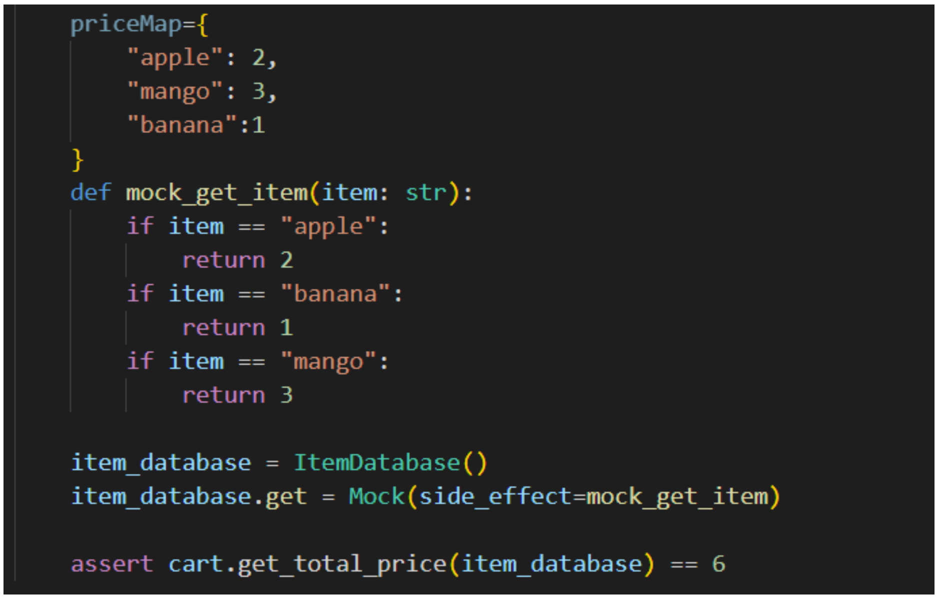
Mocks are objects that replace real objects during testing. They are used to simulate the behavior of objects that are difficult or impossible to create in a test environment. For example, mocks can be used to simulate network connections, database queries, or user input.

Mocks can be created manually using Python's unittest.mock library or can be generated automatically using Pytest's pytest-mock plugin. The pytest-mock plugin provides a mocker fixture that can be used to generate mocks for test functions. The mocker fixture can be used to replace functions, classes, or modules with mocks that simulate their behavior.

For example, if a test function requires a network connection, a mock can be created using the mocker fixture that simulates the behavior of the network connection. This ensures that the test function can be run without requiring an actual network connection and allows developers to test their code in a controlled environment.



```
def test_total_price_calculation(cart):
    cart.add("apple")
    cart.add("banana")
    cart.add("mango")
```

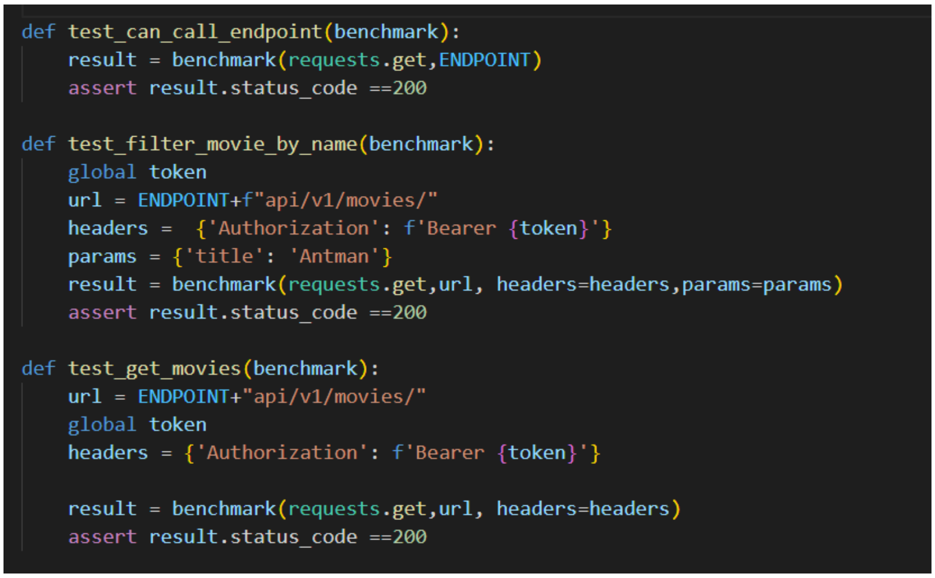


```python
    priceMap={
        "apple": 2,
        "mango": 3,
        "banana":1
    }
    def mock_get_item(item: str):
        if item == "apple":
            return 2
        if item == "banana":
            return 1
        if item == "mango":
            return 3

    item_database = ItemDatabase()
    item_database.get = Mock(side_effect=mock_get_item)

    assert cart.get_total_price(item_database) == 6
```

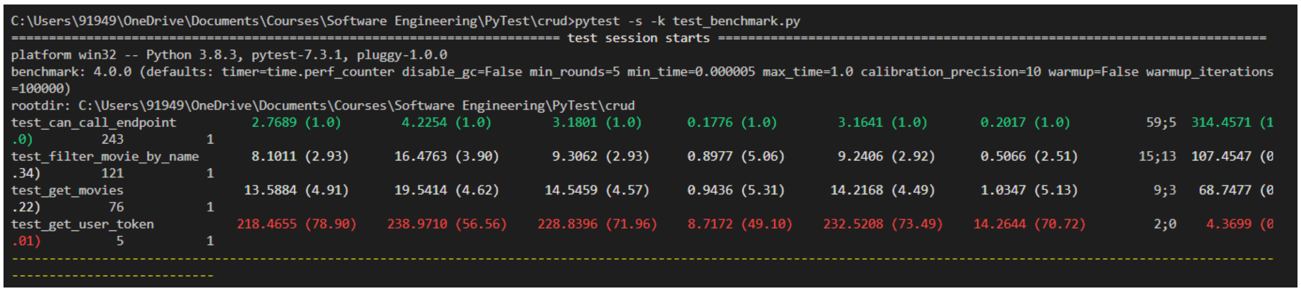
Plugins:

Pytest has a large and growing ecosystem of plugins that add additional functionality and integrations. Plugins are available for things like test coverage reporting, benchmarking, and test parallelization.

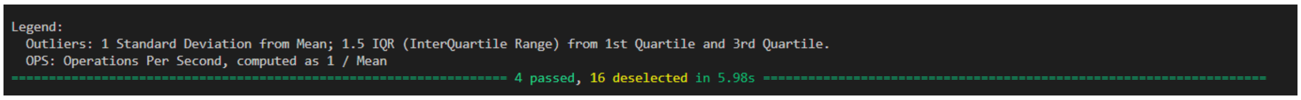
Ex: Result of BenchMark Plugin



```python
def test_can_call_endpoint(benchmark):
    result = benchmark(requests.get,ENDPOINT)
    assert result.status_code ==200

def test_filter_movie_by_name(benchmark):
    global token
    url = ENDPOINT+f"api/v1/movies/"
    headers =  {'Authorization': f'Bearer {token}'}
    params = {'title': 'Antman'}
    result = benchmark(requests.get,url, headers=headers,params=params)
    assert result.status_code ==200

def test_get_movies(benchmark):
    url = ENDPOINT+"api/v1/movies/"
    global token
    headers = {'Authorization': f'Bearer {token}'}

    result = benchmark(requests.get,url, headers=headers)
    assert result.status_code ==200
```



```
C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud>pytest -s -k test_benchmark.py
========================================================================== test session starts ==========================================================================
platform win32 -- Python 3.8.3, pytest-7.3.1, pluggy-1.0.0
benchmark: 4.0.0 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=False warmup_iterations
=100000)
rootdir: C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud
test_can_call_endpoint         2.7689 (1.0)        4.2254 (1.0)        3.1801 (1.0)       0.1776 (1.0)        3.1641 (1.0)        0.2017 (1.0)          59;5  314.4571 (1
.0)         243            1
test_filter_movie_by_name      8.1011 (2.93)      16.4763 (3.90)       9.3062 (2.93)      0.8977 (5.06)       9.2406 (2.92)       0.5066 (2.51)        15;13  107.4547 (0
.34)        121            1
test_get_movies               13.5884 (4.91)      19.5414 (4.62)      14.5459 (4.57)      0.9436 (5.31)      14.2168 (4.49)       1.0347 (5.13)          9;3   68.7477 (0
.22)         76            1
test_get_user_token          218.4655 (78.90)    238.9710 (56.56)    228.8396 (71.96)     8.7172 (49.10)    232.5208 (73.49)     14.2644 (70.72)         2;0    4.3699 (0
.01)          5            1
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
---------------------------
```



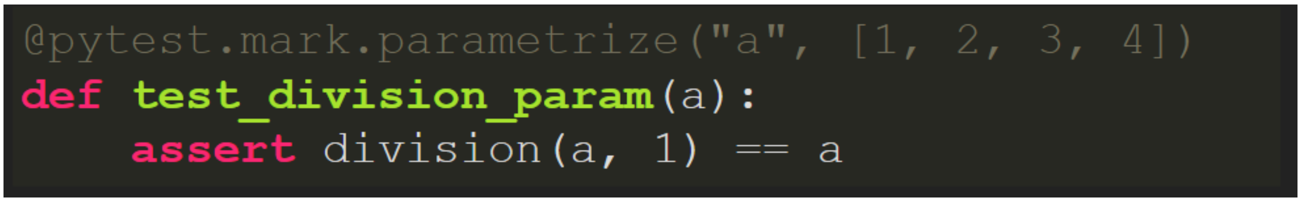
```
Legend:
  Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quartile.
  OPS: Operations Per Second, computed as 1 / Mean
================================================================ 4 passed, 16 deselected in 5.98s ================================================================
```

Assertions:

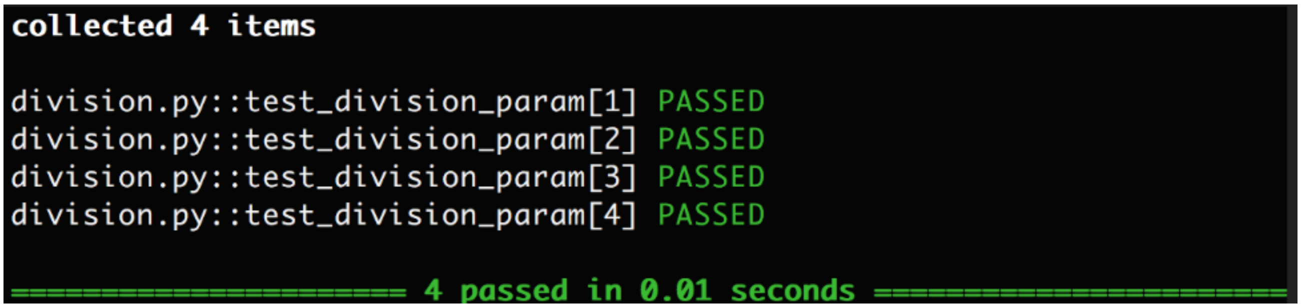
Pytest provides a comprehensive set of assertion helpers that make it easy to test complex data structures and conditions. The assertion helpers provide informative error messages that make it easy to diagnose and fix issues when tests fail.

Parameterization:

Pytest allows you to parameterize test functions, so that they can be run with multiple sets of input data. This makes it easy to test functions with different inputs and ensure that they work correctly under a range of conditions.



```python
@pytest.mark.parametrize("a", [1, 2, 3, 4])
def test_division_param(a):
    assert division(a, 1) == a
```



```
collected 4 items

division.py::test_division_param[1] PASSED
division.py::test_division_param[2] PASSED
division.py::test_division_param[3] PASSED
division.py::test_division_param[4] PASSED

====================== 4 passed in 0.01 seconds ======================
```

The weakness and missing features that not included in Pytest:

While Pytest is a powerful and flexible testing framework, there are some weaknesses and missing features that are not included out of the box. Here are a few examples:

Slow startup time: Pytest's test discovery and setup process can be relatively slow for large test suites or complex projects. This can result in slow test execution times, especially when running tests in a continuous integration (CI) environment.

Lack of built-in test data management: Pytest does not provide a built-in solution for managing test data, such as generating randomized data or setting up test fixtures. This can make it difficult to write effective and maintainable tests, especially for complex or data-intensive applications.

Limited support for distributed testing: While the pytest-xdist plugin provides some support for distributed testing across multiple CPUs or machines, Pytest does not have built-in support for distributed testing. This can make it difficult to scale test suites for large projects or distributed systems.

Limited support for parameterized testing: While Pytest's parameterization feature is powerful, it is somewhat limited in scope and can be difficult to use for more complex test scenarios. For example, it does not support dynamic parameter generation or parameterization based on external data sources.

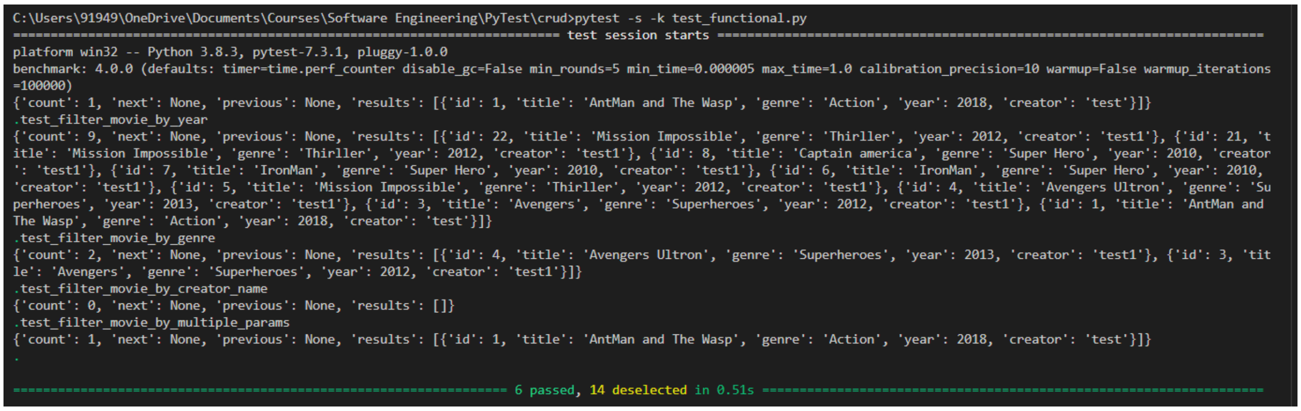
Limited support for test isolation: While Pytest provides support for fixtures and test doubles, it does not have built-in support for test isolation or sandboxing. This can make it difficult to write tests that do not interfere with each other or external systems.

Overall, while Pytest is a powerful and flexible testing framework, there are some weaknesses and missing features that can make it difficult to use for certain types of projects or scenarios. However, many of these limitations can be overcome with the help of third-party plugins or libraries, or by using alternative testing frameworks or tools.

Testing Reports

Pytest provides several options for generating test reports, including command-line output, JUnit XML, and HTML reports. Each report type has its own advantages and effectiveness, depending on the use case and intended audience.

Command-line output: Pytest provides detailed output to the console during test execution. This output includes information about each test case, including its status, execution time, and any errors or failures that occurred. This report type is most useful for developers and testers who are running tests locally or on a command-line interface.



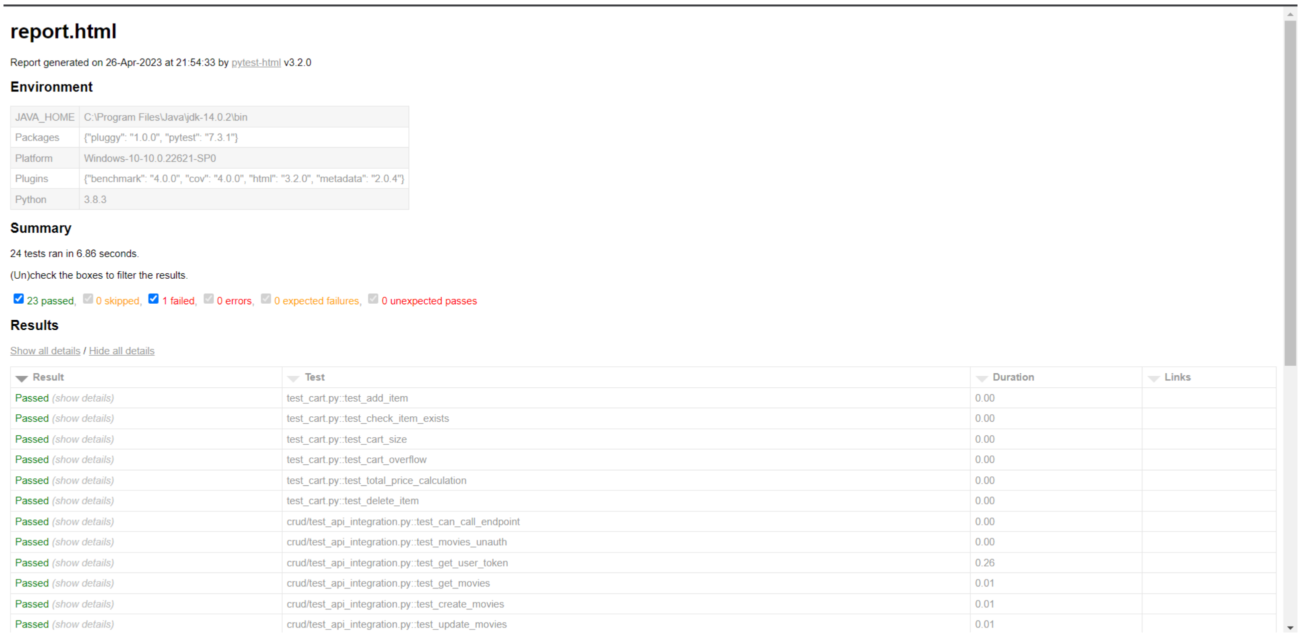
```
C:\Users\91949\OneDrive\Documents\Courses\Software Engineering\PyTest\crud>pytest -s -k test_functional.py
======================================================================== test session starts =========================================================================
platform win32 -- Python 3.8.3, pytest-7.3.1, pluggy-1.0.0
benchmark: 4.0.0 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=False warmup_iterations
=100000)
{'count': 1, 'next': None, 'previous': None, 'results': [{'id': 1, 'title': 'AntMan and The Wasp', 'genre': 'Action', 'year': 2018, 'creator': 'test'}]}
.test_filter_movie_by_year
{'count': 9, 'next': None, 'previous': None, 'results': [{'id': 22, 'title': 'Mission Impossible', 'genre': 'Thirller', 'year': 2012, 'creator': 'test1'}, {'id': 21, 't
itle': 'Mission Impossible', 'genre': 'Thirller', 'year': 2012, 'creator': 'test1'}, {'id': 8, 'title': 'Captain america', 'genre': 'Super Hero', 'year': 2010, 'creator
': 'test1'}, {'id': 7, 'title': 'IronMan', 'genre': 'Super Hero', 'year': 2010, 'creator': 'test1'}, {'id': 6, 'title': 'IronMan', 'genre': 'Super Hero', 'year': 2010,
'creator': 'test1'}, {'id': 5, 'title': 'Mission Impossible', 'genre': 'Thirller', 'year': 2012, 'creator': 'test1'}, {'id': 4, 'title': 'Avengers Ultron', 'genre': 'Su
perheroes', 'year': 2013, 'creator': 'test1'}, {'id': 3, 'title': 'Avengers', 'genre': 'Superheroes', 'year': 2012, 'creator': 'test1'}, {'id': 1, 'title': 'AntMan and
The Wasp', 'genre': 'Action', 'year': 2018, 'creator': 'test'}]}
.test_filter_movie_by_genre
{'count': 2, 'next': None, 'previous': None, 'results': [{'id': 4, 'title': 'Avengers Ultron', 'genre': 'Superheroes', 'year': 2013, 'creator': 'test1'}, {'id': 3, 'tit
le': 'Avengers', 'genre': 'Superheroes', 'year': 2012, 'creator': 'test1'}]}
.test_filter_movie_by_creator_name
{'count': 0, 'next': None, 'previous': None, 'results': []}
.test_filter_movie_by_multiple_params
{'count': 1, 'next': None, 'previous': None, 'results': [{'id': 1, 'title': 'AntMan and The Wasp', 'genre': 'Action', 'year': 2018, 'creator': 'test'}]}
.

================================================================= 6 passed, 14 deselected in 0.51s ==================================================================
```

JUnit XML: Pytest can generate JUnit-compatible XML output, which can be used with continuous integration (CI) tools and other test management systems. This report type includes detailed information about each test case, including its status, execution time, and any errors or failures that occurred. This report type is most useful for integrating Pytest with other testing and reporting tools.

HTML report: Pytest can also generate HTML reports, which provide a detailed summary of test results in a more user-friendly format. These reports include information about the test suite, individual test cases, and any failures or errors that occurred. This report type is most useful for sharing test results with stakeholders, such as project managers or business analysts.

The effectiveness of each report type depends on the intended audience and use case. Command-line output is most effective for developers and testers who are running tests locally and need detailed information about test execution. JUnit XML is most effective for integrating Pytest with other testing and reporting tools, while HTML reports are most effective for sharing test results with stakeholders who may not be familiar with testing terminology.

Overall, Pytest's reporting capabilities are flexible and can be tailored to the specific needs of a given project or organization. By using the appropriate report types and tools, testers and developers can effectively communicate test results and collaborate more effectively to ensure the quality and reliability of their software.



report.html

Report generated on 26-Apr-2023 at 21:54:33 by pytest-html v3.2.0

Environment

JAVA_HOME	C:\Program Files\Java\jdk-14.0.2\bin
Packages	{"pluggy": "1.0.0", "pytest": "7.3.1"}
Platform	Windows-10-10.0.22621-SP0
Plugins	{"benchmark": "4.0.0", "cov": "4.0.0", "html": "3.2.0", "metadata": "2.0.4"}
Python	3.8.3

Summary

24 tests ran in 6.86 seconds.

(Un)check the boxes to filter the results.

☑ 23 passed, ☐ 0 skipped, ☑ 1 failed, ☐ 0 errors, ☐ 0 expected failures, ☐ 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration	Links
Passed (show details)	test_cart.py::test_add_item	0.00	
Passed (show details)	test_cart.py::test_check_item_exists	0.00	
Passed (show details)	test_cart.py::test_cart_size	0.00	
Passed (show details)	test_cart.py::test_cart_overflow	0.00	
Passed (show details)	test_cart.py::test_total_price_calculation	0.00	
Passed (show details)	test_cart.py::test_delete_item	0.00	
Passed (show details)	crud/test_api_integration.py::test_can_call_endpoint	0.00	
Passed (show details)	crud/test_api_integration.py::test_movies_unauth	0.00	
Passed (show details)	crud/test_api_integration.py::test_get_user_token	0.26	
Passed (show details)	crud/test_api_integration.py::test_get_movies	0.01	
Passed (show details)	crud/test_api_integration.py::test_create_movies	0.01	
Passed (show details)	crud/test_api_integration.py::test_update_movies	0.01	

Conclusion

Pytest is a popular testing framework for Python that allows developers to write tests in a simple and efficient manner. It provides a powerful set of features such as test discovery, fixtures, and parameterization, that enable developers to write comprehensive tests with minimal boilerplate code.

Some of the key advantages of using Pytest are:

Simplicity: Pytest has a simple and intuitive syntax that allows developers to write tests quickly and easily. It requires minimal boilerplate code and provides a rich set of assertions that make test writing a breeze.

Flexibility: Pytest is highly flexible and can be easily integrated with other Python libraries and frameworks. It supports a wide range of plugins and extensions that enable developers to customize the testing environment to suit their specific needs.

Speed: Pytest is fast and efficient. It uses a parallel testing approach that runs tests in parallel, reducing the overall testing time.

Robustness: Pytest is a robust testing framework that handles errors and exceptions gracefully. It provides detailed error messages that help developers identify and fix issues quickly.

Overall, Pytest is a powerful testing framework that can help developers write comprehensive tests with minimal effort. Its simplicity, flexibility, speed, and robustness make it a popular choice for Python developers who want to ensure the quality and reliability of their code.

Contribution

Team Member	Contribution
Chenchu Siva Koushik Vemula	Integration testing , Django Application,Project Report
Jonnala Nihal	Unit test,Project Report
Sai Nikitha Ponnathota	Installation and setup,Project Report
Teja kamesh Gattu	Performance testing and Project Report
Javali Subha Mapati	Fuctional testing,Presentation
Nikhila Mannem	Presentation, Unit Test
Naresh Bakaram	Operational acceptance testing
Bhavana Varma Penmetsa	plugins and presentation