



Write test cases fastapi





need to write test cases a for python fastapi project
use mockito
write test cases for controller
write test cases for service

give code and configuration from scratch all step by step

Here's a complete **step-by-step guide** to write **test cases** for a Python **FastAPI project** using **Mockito** (via `mockito-python`), covering both:

-  Controller (API route) testing
-  Service layer testing

We'll build everything **from scratch**, including:

1.  Project Structure
2.  Dependencies
3.  FastAPI App with Service and Controller
4.  Writing Unit Tests for Controller and Service using `pytest` and `mockito`

1. Project Structure

markdown

```
fastapi-mockito-demo/  
|  
|—— app/  
|   |—— __init__.py  
|   |—— main.py  
|   |—— controllers.py  
|   |—— services.py  
|  
|—— tests/  
|   |—— __init__.py  
|   |—— test_controllers.py  
|   |—— test_services.py  
|
```

```
|— requirements.txt
|— pytest.ini
```

✓ 2. Dependencies

Add the following to `requirements.txt` :

```
txt

fastapi
uvicorn
pytest
httpx
mockito
```

Then install:

```
bash

pip install -r requirements.txt
```

Create `pytest.ini` :

```
ini

# pytest.ini
[pytest]
addopts = --tb=short -v
```

✓ 3. App Code

◆ `app/services.py`

```
python

# app/services.py

class MathService:
    def add(self, a: int, b: int) -> int:
        return a + b
```

```
def subtract(self, a: int, b: int) -> int:
    return a - b
```

◆ `app/controllers.py`

python

app/controllers.py

```
from fastapi import APIRouter, Depends
from app.services import MathService
```

```
router = APIRouter()
```

```
def get_math_service():
    return MathService()
```

```
@router.get("/add")
```

```
def add_numbers(a: int, b: int, service: MathService = Depends(get_math_service)):
    return {"result": service.add(a, b)}
```

```
@router.get("/subtract")
```

```
def subtract_numbers(a: int, b: int, service: MathService = Depends(get_math_service)):
    return {"result": service.subtract(a, b)}
```

◆ `app/main.py`

python

app/main.py

```
from fastapi import FastAPI
from app.controllers import router
```

```
app = FastAPI()
app.include_router(router)
```

Run the server:

```
bash
```

```
uvicorn app.main:app --reload
```

✓ 4. Test Setup

♦ `tests/test_services.py` (Service Layer)

```
python
```

```
# tests/test_services.py
```

```
import pytest
from app.services import MathService

@pytest.fixture
def math_service():
    return MathService()

def test_add(math_service):
    assert math_service.add(2, 3) == 5

def test_subtract(math_service):
    assert math_service.subtract(5, 3) == 2
```

♦ `tests/test_controllers.py` (Controller Layer)

```
python
```

```
# tests/test_controllers.py
```

```
import pytest
from fastapi.testclient import TestClient
from app.main import app
from mockito import mock, when, unstub
from app.services import MathService
from app.controllers import get_math_service

client = TestClient(app)
```

```

@pytest.fixture(autouse=True)
def unstub_after_test():
    yield
    unstub()

def test_add_controller():
    mock_service = mock(MathService)
    when(mock_service).add(10, 5).thenReturn(15)

    app.dependency_overrides[get_math_service] = lambda: mock_service

    response = client.get("/add?a=10&b=5")
    assert response.status_code == 200
    assert response.json() == {"result": 15}

def test_subtract_controller():
    mock_service = mock(MathService)
    when(mock_service).subtract(10, 3).thenReturn(7)

    app.dependency_overrides[get_math_service] = lambda: mock_service

    response = client.get("/subtract?a=10&b=3")
    assert response.status_code == 200
    assert response.json() == {"result": 7}

```

✓ 5. Run Tests

bash

pytest

✓ 6. Summary

Layer	Tool	Notes
Controller	TestClient	Test FastAPI routes
Service	pytest	Direct function testing

Layer	Tool	Notes
Mocks	<code>mockito</code>	Mock service for controller testing

🧩 Optional: Add More Complex Services?

Let me know if you want:

- DB interaction with SQLAlchemy and mock DB
- Auth flow and mocked JWT
- Dependency override for complex services
- Async service and route testing

I can expand this to real-world microservices setups too.

Would you like me to add more test types (integration, exception, edge cases)?

what is
[pytest]
addopts = --tb=short -v

The section:

```
ini

[pytest]
addopts = --tb=short -v
```

is part of the `pytest.ini` configuration file. Here's what each option means:

◆ `[pytest]`

- This declares that the following options are for pytest configuration.

◆ `addopts =`

- This lets you pass **default command-line options** to `pytest` every time you run `pytest` from the command line, so you don't have to type them manually.
-

◆ `--tb=short`

- `tb` stands for **traceback**.
- `--tb=short` tells pytest to show a **shortened traceback** when a test fails.

👉 Instead of displaying full stack traces (which can be long), it gives a **concise error summary**.

✅ Useful for **quickly identifying test failures** without too much clutter.

◆ `-v`

- This is **verbose mode**.
 - It shows **each test function name** and its result (PASSED, FAILED, etc.) in more detail.
-

✅ **Example Output with** `-v --tb=short`

```
bash

$ pytest

tests/test_services.py::test_add PASSED
tests/test_services.py::test_subtract PASSED
tests/test_controllers.py::test_add_controller PASSED
tests/test_controllers.py::test_subtract_controller PASSED
```

🧠 Summary

Option	Purpose
<code>--tb=short</code>	Shortens traceback output

Option	Purpose
<code>-v</code>	Verbose output (shows all test names)
<code>addopts = ...</code>	Makes these default every run

If you'd prefer full tracebacks or less verbosity, you can change or remove these options.

Would you like other useful `pytest.ini` configs (like coverage, markers, etc.)?

what is best solution to use `@patch` or `mock`

Great question!

Both `@patch` (from `unittest.mock`) and `mock()` (from `mockito`) are powerful mocking tools in Python. Which one is “best” depends on:

Use Case	You want
✓ Standard in Python	Use <code>@patch</code> (<code>unittest.mock</code>)
✓ Cleaner syntax, Java-style mocking	Use <code>mockito</code>
✓ More readable for simple patches	<code>mockito</code> or <code>@patch</code> , depending on familiarity
✓ Complex mocking like side effects, call args inspection	Both can do it, but <code>@patch</code> has more community support

✓ **Recommendation (Best Practice)**

Situation	Recommended Tool	Why
You are in a Python-only environment	<code>unittest.mock</code> / <code>@patch</code>	It's built-in and standard
You want Java-style mocking in Python	<code>mockito</code>	More declarative & readable
Working in team with pytest style	<code>mockito</code> or <code>@patch</code>	Depends on what's cleaner for you

✓ 1. Using `@patch` (from `unittest.mock`)

```
python

from unittest.mock import patch

@patch('app.services.MathService.add')
def test_add_controller(mock_add):
    mock_add.return_value = 10

    response = client.get("/add?a=4&b=6")
    assert response.status_code == 200
    assert response.json() == {"result": 10}
```

✓ 2. Using `mockito`

```
python

from mockito import mock, when
from app.services import MathService

def test_add_controller():
    service_mock = mock(MathService)
    when(service_mock).add(4, 6).thenReturn(10)

    app.dependency_overrides[get_math_service] = lambda: service_mock
```

```
response = client.get("/add?a=4&b=6")
assert response.status_code == 200
assert response.json() == {"result": 10}
```

✓ Final Advice

If you're using...	Use this
<code>unittest</code> -style tests	<code>@patch</code>
<code>pytest</code> with fixtures	<code>mockito</code>
Prefer built-in modules	<code>@patch</code>
Prefer readable stubbing	<code>mockito</code>

✓ My Verdict (Balanced)

- For **FastAPI + pytest**, I recommend `mockito` for:
 - Cleaner syntax (no import path gymnastics)
 - Java-style when-then semantics
 - Works well with dependency injection

However, if your team is more Pythonic and uses `unittest.mock` heavily, go with `@patch`.

Want a side-by-side comparison for both in a real test file?

give complete all decoratives in pytest

In **pytest**, **decorators** are used to mark, configure, or modify test functions or classes. Below is a **complete list of commonly used** `pytest` **decorators** with explanations and examples.

✓ 1. `@pytest.mark.<marker_name>`

Use this to **label** or **categorize** tests.

◆ Example:

```
python

import pytest

@pytest.mark.slow
def test_heavy_operation():
    ...
```

Then you can run:

```
bash

pytest -m slow
```

✓ You can define custom markers in `pytest.ini`:

```
ini

[pytest]
markers =
    slow: marks tests as slow
    db: marks tests that require database
```

✓ 2. `@pytest.mark.parametrize`

Use this to **run a test with multiple sets of arguments**.

◆ Example:

```
python

@pytest.mark.parametrize("a, b, expected", [
    (2, 3, 5),
    (5, 7, 12),
    (10, 1, 11)
])
```

```
def test_add(a, b, expected):  
    assert a + b == expected
```

✓ 3. `@pytest.fixture`

Defines reusable setup/teardown logic.

◆ Example:

```
python  
  
@pytest.fixture  
def sample_data():  
    return {"username": "john", "password": "secret"}  
  
def test_login(sample_data):  
    assert sample_data["username"] == "john"
```

You can also set **scope**:

```
python  
  
@pytest.fixture(scope="module")  
def db_conn():  
    ...
```

✓ Scopes: `function`, `class`, `module`, `package`, `session`

✓ 4. `@pytest.mark.skip`

Skip a test unconditionally.

◆ Example:

```
python  
  
@pytest.mark.skip(reason="Feature not ready")  
def test_feature():  
    ...
```

✓ 5. `@pytest.mark.skipif`

Conditionally skip a test.

◆ Example:

```
python

import sys

@pytest.mark.skipif(sys.platform == "win32", reason="Windows not supported")
def test_linux_only():
    ...
```

✓ 6. `@pytest.mark.xfail`

Mark a test as expected to fail (e.g., known bug).

◆ Example:

```
python

@pytest.mark.xfail(reason="Known bug #123")
def test_bug():
    assert 1 == 2
```

✓ `strict=True` will fail if the test passes unexpectedly.

✓ 7. `@pytest.mark.usefixtures`

Use one or more fixtures in a class or function without explicitly calling them as arguments.

◆ Example:

```
python

@pytest.mark.usefixtures("sample_data")
def test_feature():
    ...
```

✓ 8. `@pytest.fixture(autouse=True)`

Automatically apply fixture to all tests in scope.

◆ Example:

python

```
@pytest.fixture(autouse=True)
def setup_env():
    print("Auto setup for each test")
```

✓ 9. `@pytest.mark.filterwarnings`

Control warnings in specific tests.

◆ Example:

python

```
@pytest.mark.filterwarnings("ignore:*UselessWarning.*")
def test_ignore_warning():
    ...
```

✓ 10. `@pytest.mark.asyncio`

(If using `pytest-asyncio`) allows testing of async functions.

python

```
import pytest
import asyncio

@pytest.mark.asyncio
async def test_async_func():
    await asyncio.sleep(0.1)
    assert True
```

✓ Summary Table of Decorators

Decorator	Purpose
<code>@pytest.mark. <marker></code>	Tag or categorize tests
<code>@pytest.mark.parametrize</code>	Run test with multiple data sets
<code>@pytest.fixture</code>	Setup test resources
<code>@pytest.mark.skip</code>	Skip a test unconditionally
<code>@pytest.mark.skipif</code>	Conditionally skip a test
<code>@pytest.mark.xfail</code>	Mark test as expected to fail
<code>@pytest.mark.usefixtures</code>	Use fixture without calling it
<code>@pytest.fixture(autouse=True)</code>	Auto-use fixture
<code>@pytest.mark.filterwarnings</code>	Control test warnings
<code>@pytest.mark.asyncio</code>	Run async test functions (needs plugin)

✓ Bonus: CLI Help to List Markers

```
bash
```

```
pytest --markers
```

Would you like an example project showing all these in action or a cheat sheet PDF format?

give all decoratives example in one file for controller, don't skip any code, give code in chat

Sure! Below is a **complete FastAPI test file** that includes **all major** `pytest` **decorators** applied to **controller (route) tests** using `mockito`, `TestClient`, and FastAPI's dependency injection.

✓ Complete Example: `tests/test_controllers.py`

python

```
import sys
import pytest
from fastapi.testclient import TestClient
from mockito import when, mock, unstub
from app.main import app
from app.services import MathService
from app.controllers import get_math_service
```

```
client = TestClient(app)
```

```
# ✓ 1. fixture for dependency injection
```

```
@pytest.fixture
```

```
def mock_math_service():
```

```
    service = mock(MathService)
```

```
    yield service
```

```
    unstub()
```

```
# ✓ 2. fixture with autouse
```

```
@pytest.fixture(autouse=True)
```

```
def auto_cleanup():
```

```
    # auto fixture runs before/after every test
```

```
    yield
```

```
    unstub()
```

```
# ✓ 3. skip a test unconditionally
```



```

@pytest.mark.skip(reason="Skipping temporarily for demo")
def test_skip_example():
    assert False

# ✓ 4. skip a test conditionally
@pytest.mark.skipif(sys.platform == "win32", reason="Skip on Windows")
def test_skipif_example():
    assert True

# ✓ 5. expected failure
@pytest.mark.xfail(reason="Known issue: division not implemented")
def test_expected_fail():
    assert 1 / 0 == 1

# ✓ 6. filter warnings
@pytest.mark.filterwarnings("ignore::DeprecationWarning")
def test_ignore_warning():
    import warnings
    warnings.warn("This is deprecated", DeprecationWarning)
    assert True

# ✓ 7. parametrize to test multiple inputs
@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),
    (5, 5, 10),
    (10, -2, 8),
])
def test_add_parametrized(a, b, expected, mock_math_service):
    when(mock_math_service).add(a, b).thenReturn(expected)
    app.dependency_overrides[get_math_service] = lambda: mock_math_service
    response = client.get(f"/add?a={a}&b={b}")
    assert response.status_code == 200
    assert response.json() == {"result": expected}

# ✓ 8. usefixtures applies fixtures without using as arguments
@pytest.mark.usefixtures("mock_math_service")
def test_subtract_static():
    service = mock(MathService)
    when(service).subtract(8, 3).thenReturn(5)
    app.dependency_overrides[get_math_service] = lambda: service
    response = client.get("/subtract?a=8&b=3")
    assert response.status_code == 200

```

```
assert response.json() == {"result": 5}
```

 9. custom marker (must be defined in pytest.ini)

```
@pytest.mark.slow
```

```
def test_slow_case(mock_math_service):
```

```
    when(mock_math_service).add(100, 200).thenReturn(300)
```

```
    app.dependency_overrides[get_math_service] = lambda: mock_math_service
```

```
    response = client.get("/add?a=100&b=200")
```

```
    assert response.status_code == 200
```

```
    assert response.json() == {"result": 300}
```

 10. async test (optional, if using pytest-asyncio)

Requires: pip install pytest-asyncio

Uncomment if testing async route

```
"""
```

```
import asyncio
```

```
@pytest.mark.asyncio
```

```
async def test_async_example():
```

```
    await asyncio.sleep(0.1)
```

```
    assert True
```

```
"""
```

Notes

- All decorators are included: `skip`, `skipif`, `xfail`, `parametrize`, `usefixtures`, `fixture`, `autouse`, `filterwarnings`, `custom marker`, and `asyncio`.
- You need to add this to your `pytest.ini` to define the custom marker `slow`:

```
ini
```

```
[pytest]
```

```
markers =
```

```
    slow: marks tests as slow (deselect with '-m "not slow"')
```

Run Tests

```
bash
```

✓ Output Example

```
bash
```

```
tests/test_controllers.py::test_add_parametrized[1-2-3] PASSED
tests/test_controllers.py::test_add_parametrized[5-5-10] PASSED
tests/test_controllers.py::test_add_parametrized[10--2-8] PASSED
tests/test_controllers.py::test_subtract_static PASSED
tests/test_controllers.py::test_slow_case PASSED
tests/test_controllers.py::test_skip_example SKIPPED
tests/test_controllers.py::test_skipif_example SKIPPED
tests/test_controllers.py::test_expected_fail xfail
```

Let me know if you also want a **complete service-layer test** or want to **combine** `mockito` **with** `patch` examples for comparison.

hey not understanding all decoratives clearly could you please give detailed information for all decoratives

Absolutely! Let me give you a **clear, detailed explanation of each major pytest decorator**, along with use cases and examples — so you can confidently use them in your FastAPI tests or any other Python testing.

✓ 1. `@pytest.fixture`

◆ What it does:

Defines a **setup function** to provide common test data, resources, or mocks that can be reused.

◆ Syntax:

```
python
```

```
@pytest.fixture
def db_connection():
    conn = connect_to_db()
    yield conn
    conn.close()
```

◆ Use it in test:

```
python

def test_query(db_connection):
    result = db_connection.query("SELECT 1")
    assert result == 1
```

◆ Why it's useful:

- Reusable setup/teardown logic.
- Keeps your tests clean and DRY.

✓ 2. @pytest.mark.parametrize

◆ What it does:

Runs a **test multiple times** with different input values.

◆ Syntax:

```
python

@pytest.mark.parametrize("a, b, expected", [
    (2, 3, 5),
    (5, 5, 10),
    (10, -2, 8),
])

def test_add(a, b, expected):
    assert a + b == expected
```

◆ Why it's useful:

- Great for testing **different combinations** of inputs without writing repetitive tests.

✓ 3. `@pytest.mark.skip`

◆ What it does:

Skips a test **unconditionally**.

◆ Syntax:

```
python

@pytest.mark.skip(reason="Temporarily disabled")
def test_something():
    assert 1 == 2 # won't run
```

◆ Why it's useful:

- You're working on the test but don't want it to fail your test suite for now.

✓ 4. `@pytest.mark.skipif`

◆ What it does:

Skips a test **conditionally**, based on an expression.

◆ Syntax:

```
python

import sys

@pytest.mark.skipif(sys.platform == "win32", reason="Doesn't work on Windows")
def test_linux_only():
    ...
```

◆ Why it's useful:

- Useful for skipping tests in incompatible environments (OS, Python version, etc.)

✓ 5. `@pytest.mark.xfail`

◆ What it does:

Marks a test as **expected to fail** (e.g., because of a known bug).

♦ **Syntax:**

```
python

@pytest.mark.xfail(reason="Bug #123 not fixed yet")
def test_unstable_feature():
    assert 1 == 2 # will be counted as xfail, not fail
```

♦ **Why it's useful:**

- Lets you acknowledge a known failure without failing the whole suite.

✓ 6. `@pytest.mark.usefixtures`

♦ **What it does:**

Allows you to **run a fixture** without having to pass it as a parameter to the test.

♦ **Syntax:**

```
python

@pytest.fixture
def setup_user():
    print("Setting up user")

@pytest.mark.usefixtures("setup_user")
def test_profile_access():
    assert True
```

♦ **Why it's useful:**

- Clean way to run setup code even if test doesn't use the fixture value.

✓ 7. `@pytest.fixture(autouse=True)`

♦ **What it does:**

Makes the fixture **automatically applied** to every test without being referenced.

◆ Syntax:

python

```
@pytest.fixture(autouse=True)
def auto_setup():
    print("Runs before each test")
```

◆ Why it's useful:

- Apply common logic like resetting state, cleaning up, logging, etc.

✓ 8. `@pytest.mark.filterwarnings`

◆ What it does:

Allows you to **suppress or filter specific warnings** during tests.

◆ Syntax:

python

```
@pytest.mark.filterwarnings("ignore::DeprecationWarning")
def test_old_api():
    import warnings
    warnings.warn("deprecated", DeprecationWarning)
    assert True
```

◆ Why it's useful:

- Keeps test output clean from known/unavoidable warnings.

✓ 9. `@pytest.mark.asyncio`

Requires: `pip install pytest-asyncio`

◆ What it does:

Allows you to **write async tests** with `async def`.

◆ Syntax:

python

```
import pytest
import asyncio

@pytest.mark.asyncio
async def test_async_operation():
    await asyncio.sleep(0.1)
    assert True
```

◆ Why it's useful:

- Needed when testing **FastAPI async routes**, or any async functions.

✓ 10. `@pytest.mark.<custom_marker>`

◆ What it does:

Defines your own **marker/tag** for grouping tests.

◆ Syntax:

```
python

@pytest.mark.slow
def test_big_data():
    ...
```

Then in `pytest.ini`:

```
ini

[pytest]
markers =
    slow: marks tests as slow
```

Run using:

```
bash

pytest -m slow
```

◆ Why it's useful:

- Run only specific categories of tests.

✓ Bonus: `@pytest.mark.parametrize` + fixtures

You can use **fixtures together with parameterized tests**:

```
python

@pytest.fixture
def multiplier():
    return 2

@pytest.mark.parametrize("x, expected", [(1, 2), (3, 6)])
def test_times_two(x, expected, multiplier):
    assert x * multiplier == expected
```

✓ Summary Table

Decorator	Use Case
<code>@pytest.fixture</code>	Reusable setup/teardown logic
<code>@pytest.fixture(autouse=True)</code>	Automatically applies fixture
<code>@pytest.mark.parametrize</code>	Run test with multiple inputs
<code>@pytest.mark.skip</code>	Unconditionally skip test
<code>@pytest.mark.skipif</code>	Conditionally skip test
<code>@pytest.mark.xfail</code>	Mark test as expected to fail
<code>@pytest.mark.usefixtures</code>	Use a fixture without passing it

Decorator	Use Case
<code>@pytest.mark.filterwarnings</code>	Suppress specific warnings
<code>@pytest.mark.asyncio</code>	Support async test functions
<code>@pytest.mark.<custom></code>	Group/tag tests (e.g. <code>@pytest.mark.slow</code>)

Would you like this in a **cheat sheet format** or a **real test file with FastAPI routes**? I can show all decorators in context again.

explain more detailed for
`@pytest.mark.parametrize("x, expected", [(1, 2), (3, 6)])`

Sure! Let's break down `@pytest.mark.parametrize(...)` in **complete detail**, using the example:

```
python
```

```
@pytest.mark.parametrize("x, expected", [(1, 2), (3, 6)])
```

✅ What is `@pytest.mark.parametrize`?

`@pytest.mark.parametrize` is a **pytest decorator** used to run the **same test function multiple times**, each time with a **different set of inputs**.

Think of it like **data-driven testing**: You give pytest a list of inputs, and it will call your test function once for each input set.



✅ Syntax Breakdown

```
python
```

```
@pytest.mark.parametrize("x, expected", [(1, 2), (3, 6)])
def test_double(x, expected):
    assert 2 * x == expected
```

Explanation:

- `"x, expected"` → These are the **parameter names** your test function will receive.
- `[(1, 2), (3, 6)]` → This is the **list of input values** for each test run.
- So the test will run like:

Test Call	What Happens
<code>test_double(1, 2)</code>	<code>2 * 1 == 2</code> →  Pass
<code>test_double(3, 6)</code>	<code>2 * 3 == 6</code> →  Pass

How Does It Work Internally?

Pytest internally "unrolls" this into **two separate test cases**:

```
text

test_double[1-2]
test_double[3-6]
```

If one fails, it tells you **exactly which input pair** caused it.

Why It's Useful

- Avoid writing multiple similar test functions.
- Helps test **edge cases, invalid inputs, multiple scenarios** easily.
- Works with **fixtures, mocking**, and other pytest features.

Realistic Example in FastAPI Controller Testing

Imagine this route:

python

```
@app.get("/double")
def double_value(x: int):
    return {"result": x * 2}
```

Test using parameterization:

python

```
import pytest
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

@pytest.mark.parametrize("x, expected", [
    (1, 2),
    (0, 0),
    (-1, -2),
    (100, 200)
])
def test_double_route(x, expected):
    response = client.get(f"/double?x={x}")
    assert response.status_code == 200
    assert response.json() == {"result": expected}
```

✓ This will run the same test 4 times with different values of `x` and check the result.

✓ Advanced: Multiple Parameters

python

```
@pytest.mark.parametrize("a, b, expected", [
    (2, 3, 5),
    (5, 7, 12),
    (10, -1, 9),
])
def test_add(a, b, expected):
    assert a + b == expected
```

This is equivalent to:

- `test_add(2, 3, 5)`
- `test_add(5, 7, 12)`
- `test_add(10, -1, 9)`

✓ Combine with Fixtures

python

```
@pytest.fixture
def multiplier():
    return 3

@pytest.mark.parametrize("x, expected", [(1, 3), (2, 6)])
def test_custom_mult(x, expected, multiplier):
    assert x * multiplier == expected
```

✓ Error Example

python

```
@pytest.mark.parametrize("x", [1, 2, 3])
def test_missing_param(x, y):
    ...
```

✗ Will raise an error because `y` is not provided in the parametrize list.

✓ Summary Table

Component	Description
<code>"x, expected"</code>	Names of the arguments your test will accept
<code>[(1, 2), (3, 6)]</code>	List of test cases (tuples) to loop through
Test is run N times	Once per item in the list

Component	Description
Works with fixtures	Yes
Great for edge cases	Yes

Would you like a visual chart, test report example, or real integration into your FastAPI project to see this live?

how it knows what is expected value

Great question!

Let's clarify exactly **how** `@pytest.mark.parametrize(...)` **knows what the expected value is**, and why **it's not automatic** — **you explicitly define it**.

✓ Short Answer

It doesn't "know" automatically.

You, as the test writer, **provide the expected output** manually in each test case tuple like:

```
python

@pytest.mark.parametrize("x, expected", [
    (1, 2),    # x=1 → expected=2
    (3, 6),    # x=3 → expected=6
])
def test_double(x, expected):
    assert 2 * x == expected # <-- You write the logic AND expected result
```

👉 `expected` is just a **variable name** that you assign a value to in each tuple. Pytest passes that value to the test function when it runs.

✓ Behind the Scenes

Here's what happens under the hood:

Step-by-step:

1. You write:

```
python

@pytest.mark.parametrize("x, expected", [(1, 2), (3, 6)])
def test_double(x, expected):
    ...
```

2. Pytest will:

- Run `test_double(1, 2)`
- Run `test_double(3, 6)`

3. In the test, you compare your **function's output** against `expected` :

```
python

assert 2 * x == expected
```

4. If your function (or route) returns something **not equal to** `expected` , the test fails.

? Why Not Calculate Expected Automatically?

Because pytest is a **test runner**, not your **business logic** engine.

- It doesn't know your logic (e.g., `2 * x` , `x ** 2` , `fibonacci(x)` , etc.)
- You must **explicitly tell** pytest: "This input **should** produce this output."

✓ Example: Add Two Numbers

Imagine your logic is `a + b` , you write:

```
python

@pytest.mark.parametrize("a, b, expected", [
    (1, 2, 3),
    (5, 5, 10),
    (10, -2, 8),
])
```

```
def test_add(a, b, expected):  
    assert a + b == expected
```

Here, **you define:**

- $a = 1, b = 2 \rightarrow \text{expected} = 3$
- $a = 5, b = 5 \rightarrow \text{expected} = 10$
- $a = 10, b = -2 \rightarrow \text{expected} = 8$

Pytest passes each tuple to your test function.

✓ Real FastAPI Example

python

```
# your route logic  
@app.get("/double")  
def double(x: int):  
    return {"result": x * 2}
```

Test:

python

```
@pytest.mark.parametrize("x, expected", [  
    (2, 4),    # 2*2 = 4  
    (5, 10),   # 5*2 = 10  
)  
]  
def test_double_route(x, expected):  
    response = client.get(f"/double?x={x}")  
    assert response.status_code == 200  
    assert response.json() == {"result": expected}
```

Again, `expected` is your **manual assertion of what's correct.**

✓ Summary

Concept	Explanation
<code>expected</code> value	You explicitly define it for each test case

Concept	Explanation
Pytest's role	Run your test function with the data you give
Who defines logic?	You — pytest does not infer logic or outputs
Why use <code>expected</code> ?	To compare actual vs. expected result clearly

Would you like a dynamic way to generate `expected` values in tests automatically using functions or fixtures? I can show that too.

what is mockito unstub()

Great question!

✓ What is `unstub()` in `mockito`?

`unstub()` is a function provided by the `mockito` library in Python. It is used to **clear or reset all active mocks and stubs** created with `mockito`.

✓ Purpose of `unstub()`

Mockito (Python version) maintains a **global registry** of all mocks you've created using `mock()` and all stubbing done with `when(...)`.

If you don't clean them up after a test, they can **leak into other tests**, leading to:

- False positives or false negatives
- Unintended behavior (e.g., a stub from test A affects test B)

So, to **prevent test interference**, `unstub()` clears everything.

✓ When should you call `unstub()` ?

1. At the **end of every test**, either:

- Manually
 - Or automatically using a `pytest` fixture with `autouse=True`
-

✓ Example: With and Without `unstub()`

✗ Without `unstub()` (unsafe)

python

```
from mockito import mock, when
from app.services import MathService
```

```
def test_add():
    service = mock(MathService)
    when(service).add(1, 2).thenReturn(10)
```

```
def test_add_again():
    service = mock(MathService)
    # This test may still use the stub from above (1 + 2 -> 10) incorrectly!
```

✓ With `unstub()` (safe)

python

```
import pytest
from mockito import unstub, mock, when
from app.services import MathService
```

```
@pytest.fixture(autouse=True)
def cleanup():
    yield
    unstub() # <-- clear all mocks/stubs after each test
```

```
def test_add():
    service = mock(MathService)
    when(service).add(1, 2).thenReturn(10)
```

```
def test_add_again():
    service = mock(MathService)
    when(service).add(1, 2).thenReturn(3) # fresh stub, unaffected
```

✓ Summary

Feature	<code>mockito.unstub()</code>
Purpose	Clears all mocks and stubs created globally
When to use	After each test to avoid interference
Common usage	Inside a fixture with <code>autouse=True</code>
Side effect	Makes each test independent and isolated

✓ Best Practice

Use `unstub()` **automatically** in a `pytest` fixture:

```
python

@pytest.fixture(autouse=True)
def auto_unstub():
    yield
    unstub()
```

This way, you never forget to clean up.

Let me know if you'd like a live example showing how forgetting `unstub()` can cause test failures or false results!

@pytest.fixture(autouse=True) how know it it will run before or after?

Great question! Let's break it down clearly:

✓ @pytest.fixture(autouse=True) — When does it run?

The behavior depends on where you put the `yield` in the fixture.

◆ If `yield` is in the middle:

- Everything **before** `yield` runs **before** the test.
- Everything **after** `yield` runs **after** the test.

✓ Example 1: @pytest.fixture(autouse=True) — Both before and after

python

```
import pytest

@pytest.fixture(autouse=True)
def setup_and_teardown():
    print("\n[Setup] Before test")
    yield
    print("[Teardown] After test")

def test_example():
    print("Running test!")
```

◆ Output:

bash

```
[Setup] Before test
Running test!
[Teardown] After test
```

So:

- Code **before** `yield` runs as test setup.
- Code **after** `yield` runs as test teardown.

✓ Use case for `mockito.unstub()`

```
python

from mockito import unstub
import pytest

@pytest.fixture(autouse=True)
def auto_unstub():
    yield
    unstub() # ← This runs after every test
```

👉 `unstub()` is **cleanup**, so it runs **after** the test completes.

✓ Summary

Location	Runs	Purpose
Before <code>yield</code>	Before test	Setup phase
After <code>yield</code>	After test	Teardown / cleanup

Let me know if you want a visual flowchart or timing diagram!

i no need before un anything i need only after then what to do?

Perfect — if you **only need something to run *after* each test** (like `unstub()` for cleaning up mocks), you can use `@pytest.fixture(autouse=True)` with **only the code after** `yield`.

✓ Minimal After-Only Fixture

python

```
import pytest
from mockito import import unstub

@pytest.fixture(autouse=True)
def cleanup_after_test():
    yield # nothing before the test
    unstub() # this runs *after* every test
```

◆ Explanation:

- The test runs during the `yield` pause.
- `unstub()` is called **only after** the test finishes.
- No setup code is needed — this is exactly what you want.

✓ You can use it in your `conftest.py`

If you place this in `tests/conftest.py`, it will apply to **all tests** automatically without modifying any test files.

✓ Full Example

python

```
# tests/conftest.py
import pytest
from mockito import import unstub

@pytest.fixture(autouse=True)
def auto_cleanup():
    yield
    unstub()
```

python

```
# tests/test_math.py
from mockito import import mock, when
```

```
from app.services import MathService

def test_add():
    service = mock(MathService)
    when(service).add(1, 2).thenReturn(10)
    assert service.add(1, 2) == 10 # After this, unstub() will be called
```

Let me know if you'd like to verify this works or test with multiple cases!

same i don't need after i need only before

Great! If you **only need to run code *before* each test** (e.g., setup mocks or state), and do **nothing after**, just put your logic **before the** `yield` — and leave the after part empty.

✓ Minimal Before-Only Fixture

```
python

import pytest

@pytest.fixture(autouse=True)
def setup_before_test():
    print("[Setup] Running before test")
    yield # No code after this — nothing will run after the test
```

◆ Behavior:

- Code **before** `yield` runs before each test.
- No teardown/cleanup runs after the test.

✓ Full Example

```
python
```

```
# tests/conftest.py
import pytest

@pytest.fixture(autouse=True)
def setup_before_test():
    print("[Setup] Initialize resources before each test")
    yield # Only before logic
```

python

```
# tests/test_sample.py
def test_one():
    print("Executing test one")
    assert True

def test_two():
    print("Executing test two")
    assert True
```

◆ Output:

bash

```
[Setup] Initialize resources before each test
Executing test one
[Setup] Initialize resources before each test
Executing test two
```

✓ Notes:

- You can initialize variables, mocks, environment setup, logging setup, etc.
- Just omit any code after `yield` to **skip teardown**.

Let me know if you want to inject test-specific setup values (e.g., mock service injection), or use named fixtures instead of `autouse=True` .

i have multiple test files for controllers, services and utilities how to run all test classes while building docker files in CICD

To run **all your test files** (e.g., controllers, services, utilities) during a **Docker-based CI/CD build**, follow this **step-by-step setup**:

✅ Step 1: ✅ Standard Project Structure (example)

```
your_project/
├── app/
│   ├── controllers/
│   ├── services/
│   └── utils/
├── tests/
│   ├── controllers/
│   ├── services/
│   └── utils/
├── conftest.py
├── Dockerfile
├── requirements.txt
└── pytest.ini
```

✅ Step 2: `pytest.ini` to configure discovery

Create a `pytest.ini` file in the root of the project to include custom options:

```
ini

# pytest.ini
[pytest]
addopts = --tb=short -v
testpaths = tests
python_files = test_*.py
```

This ensures:

- All test files like `test_*.py` inside `tests/**` are picked.
- Verbose output and short traceback.

✓ Step 3: Dockerfile with test execution

Update your `Dockerfile` to install dependencies and run tests during build.

Dockerfile

```
# Dockerfile
FROM python:3.11-slim

# Set working dir
WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy project files
COPY . .

# Run all tests
RUN pytest # Or: RUN pytest tests/

# Optional: start the app (only if build succeeds)
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

- ♦ `RUN pytest` will fail the build if any test fails — ideal for CI/CD.

✓ Step 4: Add to CI pipeline (GitHub Actions example)

yaml

```
# .github/workflows/test.yml
name: Test FastAPI App

on:
  push:
    branches: [main]
```

```
pull_request:
```

```
jobs:
```

```
test:
```

```
  runs-on: ubuntu-latest
```

```
steps:
```

```
- name: Checkout code
```

```
  uses: actions/checkout@v3
```

```
- name: Build Docker Image and Run Tests
```

```
  run: |
```

```
    docker build -t fastapi-app .
```

✓ CI will fail the build if any test fails.

✓ Step 5: Run manually (optional)

To run tests inside Docker manually:

```
bash
```

```
docker build -t fastapi-app .
```

Or just:

```
bash
```

```
pytest # If running outside Docker
```

✓ Summary

What	How
Discover all test files	Use <code>pytest.ini</code> with <code>testpaths=tests</code>
Run tests during Docker	<code>RUN pytest</code> in your Dockerfile

What	How
CI/CD test integration	Add Docker build step in GitHub Actions or CI tool
Fail on test error	Pytest will exit non-zero → Docker build fails

Let me know if you're using a specific CI tool (like GitHub Actions, GitLab CI, Jenkins), and I'll give you an exact setup for that!

am using jenkins

if it fails how can i check, where is report

Great — since you're using **Jenkins**, you can:

1. **Run your FastAPI tests using `pytest` in Docker** during the Jenkins pipeline.
2. **Generate a report** (textual or XML/HTML).
3. **Expose the report in Jenkins UI** using plugins.

Here's the complete Jenkins + Docker + Pytest setup, step by step 📌

✓ 1. Install Jenkins Plugins (if not already)

In Jenkins:

- Go to **Manage Jenkins** → **Plugins**.
- Make sure these are installed:
 - 🛠️ **JUnit** (for parsing XML test reports)
 - 📦 **Docker Pipeline** (if using Docker commands in pipeline)
 - 💡 (optional) **HTML Publisher** (if you want HTML reports)

✓ 2. Generate a JUnit-style test report

Update your `Dockerfile` :

```
dockerfile
```

```
# Dockerfile
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

# Run pytest and output JUnit XML report
RUN pytest --junitxml=report.xml
```

This will:

- Run all tests.
- Save an XML report file as `report.xml`.

Jenkins understands `report.xml` using the JUnit plugin.

✓ 3. Jenkins Pipeline Script (`Jenkinsfile`)

Option A: Declarative Pipeline

```
groovy

pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }

        stage('Build and Test in Docker') {
            steps {
                script {
                    docker.build('fastapi-app')
                }
            }
        }
    }
}
```

```

}

stage('Extract Report') {
  steps {
    sh 'docker create --name temp fastapi-app'
    sh 'docker cp temp:/app/report.xml ./report.xml'
    sh 'docker rm temp'
  }
}

post {
  always {
    junit 'report.xml'
  }
}
}

```

Option B: Scripted Pipeline

```

groovy

node {
  stage('Checkout') {
    checkout scm
  }

  stage('Build Docker Image and Run Tests') {
    sh 'docker build -t fastapi-app .'
  }

  stage('Copy Report') {
    sh 'docker create --name temp fastapi-app'
    sh 'docker cp temp:/app/report.xml ./report.xml'
    sh 'docker rm temp'
  }

  stage('Publish Report') {
    junit 'report.xml'
  }
}

```

✓ 4. Where is the Report Shown?

- After a Jenkins build finishes, go to the **build page**.
- Look for a section like:

✓ Test Result

Shows number of tests, passed/failed/skipped, and stack traces.

✓ 5. Optional: HTML Report (nicer look)

Use the `pytest-html` plugin:

```
bash
```

```
pip install pytest-html
```

Then change Docker test command:

```
dockerfile
```

```
RUN pytest --junitxml=report.xml --html=report.html --self-contained-html
```

And in Jenkinsfile:

```
groovy
```

```
post {  
    always {  
        junit 'report.xml'  
        publishHTML(target: [  
            reportDir: '.',  
            reportFiles: 'report.html',  
            reportName: 'Pytest HTML Report'  
        ])  
    }  
}
```

After build: see a **Pytest HTML Report** link on the Jenkins UI.

✓ Summary

Step	Tool/Action
Run tests in Docker	<code>RUN pytest --junitxml=report.xml</code>
Expose report	<code>junit 'report.xml'</code> in Jenkins
Optional nice UI report	<code>pytest-html</code> + <code>publishHTML</code> plugin
Check result on Jenkins UI	Build → Test Result / HTML Report

Would you like a complete zip-ready sample project with Jenkins + FastAPI + pytest pre-configured?

how can i know how much code coverage done in testing

To **measure code coverage** in your FastAPI project (or any Python project), you can use the `coverage.py` tool, often together with `pytest`.

✅ Step-by-Step: Measure Code Coverage

1. Install Dependencies

```
bash
```

```
pip install pytest coverage pytest-cov
```

2. Run Pytest with Coverage

Use the `--cov` option to measure coverage:

```
bash
```



```
pytest --cov=your_package_name tests/
```

Example:

```
bash
```

```
pytest --cov=app tests/
```

This will:

- Run all tests in `tests/`
- Show a coverage summary for all files inside the `app/` module

3. Generate Detailed Coverage Report

You can generate various report formats:

Terminal Report (default)

```
bash
```

```
pytest --cov=app tests/
```

Generate an HTML report (recommended):

```
bash
```

```
pytest --cov=app --cov-report=html tests/
```

After running, open:

```
bash
```

```
htmlcov/index.html
```

You'll see:

- Line-by-line source code
- Highlighted which lines were not covered

XML report for CI:

```
bash
```

```
pytest --cov=app --cov-report=xml tests/
```

Creates `coverage.xml`, useful for CI tools (Jenkins, GitHub Actions, etc.)

4. See Coverage Percentage in Terminal

Example output:

markdown

----- coverage: platform linux, python 3.11 -----

Name	Stmts	Miss	Cover

app/controllers.py	60	4	93%
app/services.py	80	10	88%
app/utils.py	30	0	100%

TOTAL	170	14	92%

✓ Configure `pytest.ini` (optional)

To make running tests easier, you can add a `pytest.ini` file:

ini

```
# pytest.ini
```

```
[pytest]
```

```
addopts = --cov=app --cov-report=term --cov-report=html:htmlcov
```

Now you can just run:

bash

```
pytest
```

And it will:

- Run coverage on `app/`
- Show terminal report
- Generate `htmlcov/index.html`

✓ Integrate with Jenkins CI

In `Dockerfile` or Jenkins pipeline:

```
bash
```

```
pytest --cov=app --cov-report=xml
```

Then in `Jenkinsfile` :

```
groovy
```

```
post {  
    always {  
        junit 'report.xml'  
        recordCoverage tools: [coberturaAdapter('coverage.xml')]  
    }  
}
```

Summary

Task	Command
Run coverage	<code>pytest --cov=app tests/</code>
Generate HTML report	<code>--cov-report=html</code> → open <code>htmlcov/index.html</code>
Generate XML report	<code>--cov-report=xml</code> → <code>coverage.xml</code>
View in Jenkins	<code>recordCoverage</code> in Jenkins pipeline

Would you like me to create a ready-to-use sample test coverage setup for your FastAPI project in a zip format or GitHub repo structure?

Yes, **SonarQube** can absolutely be used to measure **code coverage**, **code quality**, and **security analysis** for your **FastAPI Python project**, just like with other programming languages. Here's how you can set it up and integrate with your **tests** and **coverage reports**.

✅ Step-by-Step Guide: Code Coverage with SonarQube for FastAPI

🚧 1. Install SonarQube Locally (Optional for Testing)

You can run SonarQube locally using Docker:

```
bash

docker run -d --name sonarqube \
  -p 9000:9000 \
  sonarqube:lts
```

Then access: <http://localhost:9000>

Default credentials:

- **Username:** `admin`
 - **Password:** `admin`
-

📦 2. Install Required Python Tools

```
bash

pip install pytest pytest-cov coverage
```

Also, install the SonarQube scanner:

- CLI: <https://docs.sonarsource.com/sonar-scanner/>
 - or use Docker: `sonarsource/sonar-scanner-cli`
-

3. Generate Coverage Report

```
bash

pytest --cov=app --cov-report=xml
```

This will generate a file named `coverage.xml`.

4. Create `sonar-project.properties` File

In your project root, add this file:

```
properties

# sonar-project.properties

# Project Identification
sonar.projectKey=fastapi_project
sonar.projectName=FastAPI Project
sonar.projectVersion=1.0

# Source & Test Directories
sonar.sources=app
sonar.tests=tests

# File Encoding
sonar.sourceEncoding=UTF-8

# Python Coverage
sonar.python.coverage.reportPaths=coverage.xml
```

If you are using Docker, map the `coverage.xml` correctly.

5. Run SonarScanner

Using CLI (assuming installed globally):

```
bash

sonar-scanner
```

Using Docker:

bash

```
docker run --rm -e SONAR_HOST_URL="http://localhost:9000" \  
-e SONAR_LOGIN="your_sonar_token" \  
-v "$(pwd):/usr/src" \  
sonarsource/sonar-scanner-cli
```



Note: You'll need to generate a **project token** from SonarQube UI → **My Account** → **Security** → **Generate Token**



6. Check the SonarQube Dashboard

Visit: <http://localhost:9000>

You will see:

- Code coverage %
- Bugs
- Vulnerabilities
- Code smells
- Duplication
- Maintainability index



Jenkins CI Integration

You can add this to your `Jenkinsfile`:

groovy

```
pipeline {  
  agent any  
  stages {  
    stage('Test & Coverage') {  
      steps {  
        sh 'pytest --cov=app --cov-report=xml'  
      }  
    }  
    stage('SonarQube Analysis') {
```

```

environment {
    SONAR_TOKEN = credentials('sonar-token')
}
steps {
    withSonarQubeEnv('My SonarQube Server') {
        sh 'sonar-scanner'
    }
}
}
}
}
}

```

Summary

Step	Tool	Output
Run tests with coverage	<code>pytest</code> + <code>coverage</code>	<code>coverage.xml</code>
Configure SonarQube	<code>sonar-project.properties</code>	Static config
Run Sonar Scanner	<code>sonar-scanner</code>	Uploads code + coverage
View in dashboard	SonarQube	Web UI reports + graphs

Optional Enhancements

- Add `flake8`, `pylint` for linting
- Track security issues with `bandit`
- Run `black` / `isort` for formatting

Would you like a working GitHub project template with FastAPI + SonarQube + Jenkins + coverage reports preconfigured?

