# **Pytest Agenda**

## A Practical Guide

- The "Why": A Framework for Thinking About Testing
- Introduction to pytest: The De Facto Standard
- Core Concepts: Assertion, Discovery, and Configuration
- Deep Dive: Fixtures The Cornerstone of pytest
- Scaling Your Test Suite: Markers & Parametrization
- The pytest Ecosystem: Essential Plugins
- Best Practices & Integration
- Q&A

# The Testing Pyramid: A Strategic Approach

A Practical Guide

#### Unit Tests (Base)

Test individual functions or classes in isolation. They are fast, cheap, and numerous. This is pytest's primary domain.

#### Integration Tests (Middle)

Verify that different components of your application work together correctly. pytest is excellent for this.

#### End-to-End (E2E) Tests (Top)

Simulate a full user journey. They are slow, expensive, and brittle. Use them sparingly for critical paths.

Our Focus: Building a solid foundation with Unit and Integration tests using pytest.

# pytest: The De Facto Standard

A Practical Guide

pytest is more than just a test runner; it's a comprehensive framework designed for productivity and scalability.

## Why did it win the testing war?

- Minimal Boilerplate: Uses plain assert statements, making tests highly readable and Pythonic.
- **Powerful Fixture Model:** An elegant dependency injection system for managing test state and setup/teardown.
- **Rich Plugin Ecosystem:** Hundreds of plugins for everything from coverage reports (pytest-cov) to parallel execution (pytest-xdist).
- Advanced Features: Built-in support for parametrisation, markers, and detailed test failure reporting.

# **Core Concepts: Writing Effective Tests**

A Practical Guide

Let's start with a function that can fail.

```
# data_processor.py def process_data(data): if not isinstance(data, dict): raise
TypeError("Input data must be a dictionary") return data.get("value", 0) * 10 #
test_data_processor.py import pytest from data_processor import process_data def
test_process_data_raises_type_error_on_invalid_input(): with
pytest.raises(TypeError, match="Input data must be a dictionary"):
process_data("not a dict") def test_process_data_returns_correct_value(): assert
process_data({"value": 5}) == 50 def test_float_precision(): assert (0.1 + 0.2)
== pytest.approx(0.3)
```

```
To run, simply execute: $ pytest
```

# **Scaling Your Suite: Markers & Parametrization**

A Practical Guide

## 1. Markers (@pytest.mark): Tag tests to categorize them

```
@pytest.mark.slow def test_complex_calculation(): # ... a test that takes a long
time pass @pytest.mark.api def test_api_endpoint(): # ... a test that hits a live
API pass
```

```
Run only API tests: $ pytest -m api
Run all except slow: $ pytest -m "not slow"
```

## 2. Parametrization: Run one test with multiple inputs

```
@pytest.mark.parametrize("test_input, expected_output", [ ((2, 3), 5), ((-1, 1),
0), ((0, 0), 0), ((-1, -1), -2), ]) def test_add_multiple_cases(test_input,
expected_output): assert add(test_input[0], test_input[1]) == expected_output
```

# The pytest Ecosystem: Essential Plugins

A Practical Guide

Don't reinvent the wheel. Leverage the powerful plugin community.

### pytest-cov

Measures your code coverage. Essential for understanding how much of your codebase is actually being tested.

```
$ pytest --cov=my project
```

### pytest-xdist

Runs your tests in parallel across multiple CPU cores, dramatically reducing test execution time.

```
$ pytest -n auto
```

#### pytest-mock

A convenient wrapper around Python's standard unittest.mock library, provided as a fixture for easy use.

```
def test_something(mocker): ...
```

### pytest-env

Easily manage environment variables for your tests directly from your pytest.ini file.

## **Best Practices for Professional Test Suites**

A Practical Guide

#### **Tests Must Be FIRST:**

- Fast: Slow tests get ignored
- **Independent:** Tests should not depend on each other
- **Repeatable:** Same result every time, regardless of environment
- **Self-Validating:** Clear pass/fail result without manual inspection
- **Timely:** Write tests alongside or before feature code (TDD)

#### Additional Guidelines:

- Structure: Organize tests/ directory to mirror your source code structure
- Clarity: Use descriptive names like test\_login\_fails\_with\_invalid\_password
- conftest.py: Use for fixtures shared across multiple test files

## **Q&A** and Resources

A Practical Guide

## **Key Takeaway:**

pytest is a powerful, flexible framework that scales from small projects to massive enterprise applications. Mastering its features like fixtures and parametrization is key to writing efficient and maintainable tests.

## **Further Reading:**

- Official pytest Documentation: docs.pytest.org
- pytest Plugins List

## **Questions?**

# **Testing Strategies & CI Basics**

A Practical Guide for the Personal Banking Management System

## **Training Agenda**

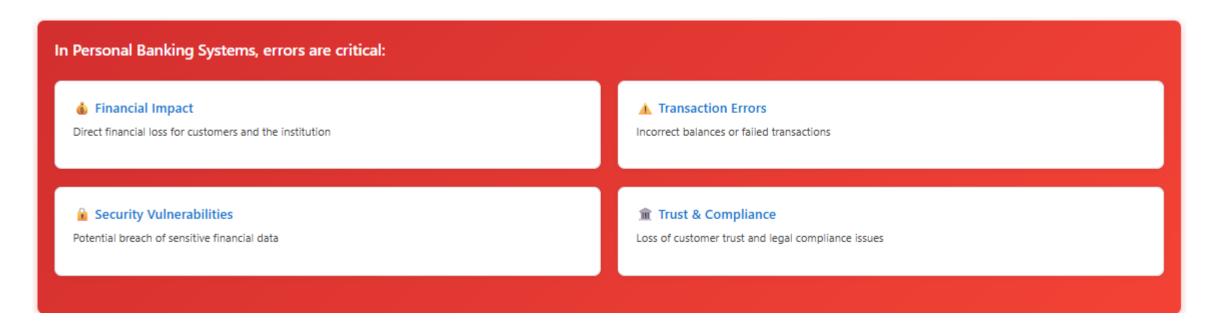
#### **Testing Foundations**

- Why Testing Matters for Banking Apps
- Unit Testing with pytest
- Writing Your First Test
- Exercise #1: Test Transactions

#### Advanced Testing & CI

- Mocking Dependencies
- Exercise #2: Mock Credit Score Check
- CI Basics with Jenkins
- Best Practices & Pro Tips

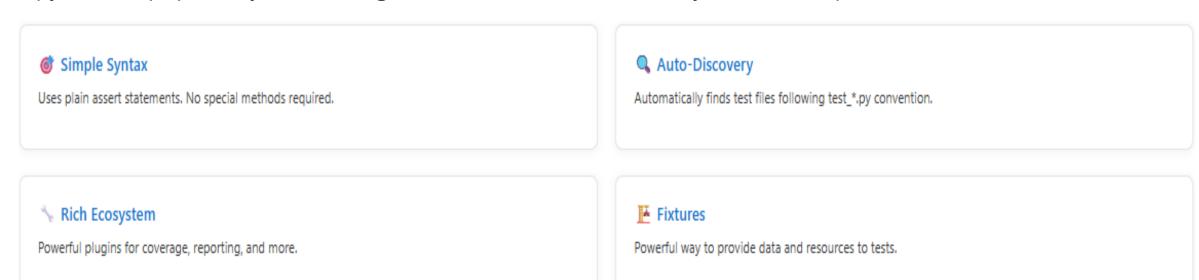
# **Why Testing Matters?**



**Automated testing** helps us build reliable and secure systems by catching issues early and ensuring new features don't break existing functionality.

# **Unit Testing with pytest**

pytest is a popular Python testing framework that makes it easy to write simple, scalable tests.



# **Writing Your First Test: Account Balance**

## **Account Class Implementation**

## banking/account.py

```
class Account:
    def __init__(self, initial_balance=0):
        if initial_balance < 0:
            raise ValueError("Initial balance cannot be negative.")
        self._balance = initial_balance

def get_balance(self):
        return self._balance

def deposit(self, amount):
    if amount <= 0:
        raise ValueError("Deposit amount must be positive.")
    self._balance += amount</pre>
```

## tests/test\_account.py

```
# Test function to check initial balance
def test_initial_balance():
    account = Account(100)
    assert account.get_balance() == 100

# Test function to check a simple deposit
def test_deposit():
    account = Account(50)
    account.deposit(50)
    assert account.get_balance() == 100
```

## **Exercise #1: Test a Transaction**

of Goal: Write a pytest test for a withdraw method

## Step 1: Add a withdraw method to the Account class

### banking/account.py

```
# Add this method to your Account class
def withdraw(self, amount):
    if amount <= 0:
        raise ValueError("Withdrawal amount must be positive.")
    if amount > self._balance:
        raise ValueError("Insufficient funds.")
    self._balance -= amount
```

## Step 2: Your Task (5 minutes)

- Create an account with initial balance of 200
- Withdraw 50
- Assert that the final balance is 150

Write function: test\_successful\_withdrawal()

## **Exercise #1: Solution**

## tests/test\_account.py

```
from banking.account import Account

# ... (previous tests) ...

def test_successful_withdrawal():
    # 1. Create an account with an initial balance of 200
    account = Account(200)

# 2. Withdraw 50
    account.withdraw(50)

# 3. Assert that the final balance is 150
    assert account.get_balance() == 150
```

#### P Bonus Test Cases

- test\_withdraw\_insufficient\_funds()
- test\_withdraw\_negative\_amount()

#### Test Structure

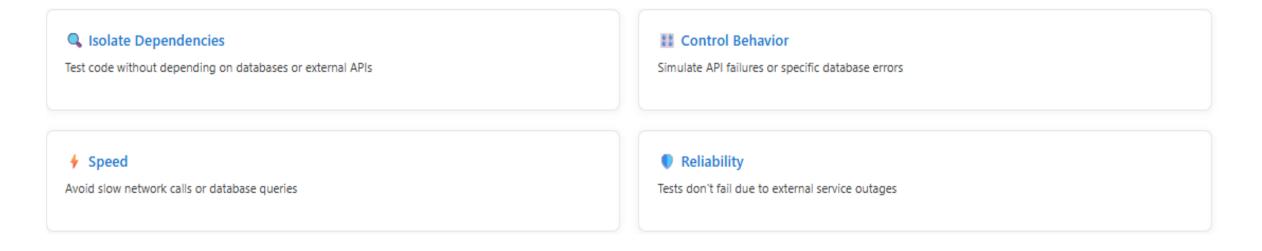
- · Arrange: Set up test conditions
- . Act: Call the method
- · Assert: Verify the outcome

# Mocking with unittest.mock

## What is Mocking?

Mocking replaces real objects with "fake" objects that simulate their behaviour.

Why do we need it in Banking Apps?



**Example: Currency Exchange API** 

We don't want tests to fail if the external exchange rate API is down!

# **Isolating External Services**

#### International Transfer Function

## banking/transfers.py

```
import requests

def get_exchange_rate(from_currency, to_currency):
    # This makes a real network call!
    response = requests.get(f"https://api.exchangeratesapi.io/latest?base={from_currency}")
    return response.json()['rates'][to_currency]

def international_transfer(from_account, to_account, amount, currency):
    rate = get_exchange_rate('USD', currency)
    converted_amount = amount * rate
    # ... logic to perform transfer ...
    return converted_amount
```

## Mocking the External Dependency

tests/test\_transfers.py

```
from unittest.mock import patch
from banking.transfers import international_transfer

# The patch decorator replaces the real function with a mock
@patch('banking.transfers.get_exchange_rate')
def test_international_transfer(mock_get_rate):
    # Configure the mock to return a predictable value
    mock_get_rate.return_value = 1.25 # 1 USD = 1.25 EUR

# Call the function under test
    converted = international_transfer(None, None, 100, 'EUR')

# Assert that our function used the mocked value correctly
    assert converted == 125.0
```

**Pro Tip:** Only mock what you own. It's better to mock your own function that *calls* the external service (get\_exchange\_rate) than to mock requests.get directly. This makes your tests less brittle.

## Exercise #2: Mock a Credit Score Check

**6** Goal: Test loan\_application function with external credit score service

## Step 1: Code to Test

## banking/loans.py

```
def check_credit_score(customer_id):
    # This function is slow and connects to an external service.
    # In a real scenario, it would make a network call.
    print(f"Checking credit score for {customer_id}...")
    # Let's pretend it returns a score.
    return 750

def process_loan_application(customer_id, amount):
    score = check_credit_score(customer_id)
    if score > 700:
        return "Approved"
    else:
        return "Rejected"
```

## Step 2: Your Task (5 minutes)

- Mock the check\_credit\_score function
- Force mock to return low score (650)
- Call process\_loan\_application
- Assert result is "Rejected"

## Exercise #2: Solution

## tests/test\_loans.py

```
from unittest.mock import patch
from banking.loans import process loan application
@patch('banking.loans.check credit score')
def test_loan_rejected_due_to_low_score(mock_check_score):
    # 1. Force the mock to return a low score
    mock check score.return value = 650
    # 2. Call the function and assert the outcome
    result = process loan application("customer-123", 5000)
    assert result == "Rejected"
# Bonus test for the "Approved" path
@patch('banking.loans.check credit score')
def test loan approved with high score(mock check score):
    mock check score.return value = 800
    result = process loan application("customer-456", 10000)
    assert result == "Approved"
```

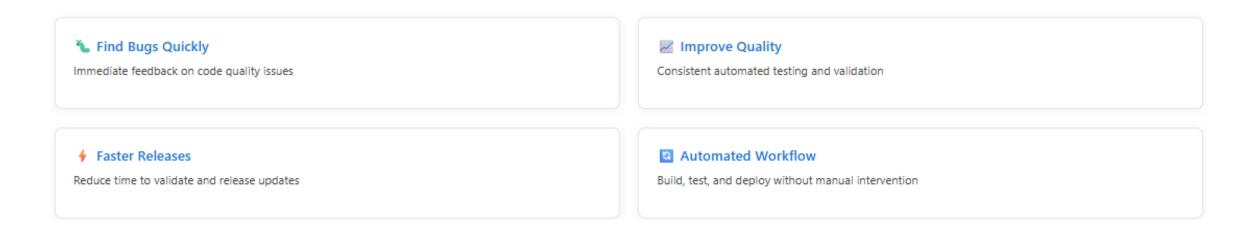
## Complete Test Coverage

We now test both approval and rejection paths without depending on external services!

## CI Basics with Jenkins

## What is Continuous Integration (CI)?

CI is a development practice where developers frequently merge code changes, triggering automated builds and tests.



**Code Commit** 

**Auto Detection** 

**Auto Detection** 

**Notification** 

Developer pushes to Git

Jenkins detects changes

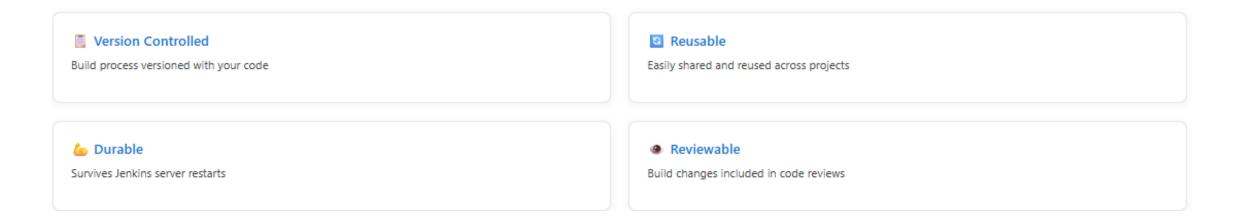
Jenkins detects changes

Team gets results

## Introduction to Jenkinsfile

## **Pipeline as Code**

A Jenkinsfile defines your build pipeline and lives alongside your source code.



## Why Pipeline as Code?

Treat your build and deployment process with the same rigor as your application code: use version control, peer review, and automated testing.

# A Simple CI Pipeline

```
pipeline {
    agent any // Run on any available Jenkins agent
    stages {
        stage('Checkout') {
            steps {
                // Get the source code from Git
                git 'https://github.com/your-repo/personal-banking.git'
        stage('Install Dependencies') {
            steps {
                // Install Python dependencies using a virtual environment
                sh 'python -m venv venv'
                sh 'source venv/bin/activate && pip install -r requirements.txt'
        stage('Run Tests') {
            steps {
                // Run our pytest tests!
                sh 'source venv/bin/activate && pytest'
        stage('Build') {
            steps {
                // In a real app, this might build a Docker image or package
                echo 'Building the application...'
                // sh './build-script.sh'
    post {
            // This block runs regardless of the pipeline's status
            echo 'Pipeline finished.'
            cleanWs() // Clean up the workspace
        success {
            echo 'Pipeline succeeded!'
        failure {
            // Send a notification if the pipeline fails
            echo 'Pipeline failed!'
            // mail to: 'team@example.com', subject: 'Build Failed'
```

# **Best Practices & Pro Tips**

### Testing Best Practices

- . Test one thing at a time: Single behavior per test
- Arrange-Act-Assert: Clear test structure
- · Descriptive names: test\_withdrawal\_with\_sufficient\_funds
- . Focus on critical paths: Test business logic first

#### CI Best Practices

- Keep builds fast: Quick feedback loops
- · Commit frequently: Small, focused changes
- . Never commit broken code: Test locally first
- . Fix breaks immediately: Top team priority

## **®** Remember: Quality is Everyone's Responsibility

In banking systems, automated testing and CI aren't optional - they're fundamental to building secure, trustworthy products.

# **Key Takeaways**

#### **©** What We've Learned

- · pytest enables clean, readable, and effective unit tests to ensure code reliability
- · unittest.mock is essential for isolating code from external dependencies
- Jenkins & Jenkinsfile automate build and test processes for rapid feedback
- . Automated testing and CI are fundamental for critical banking applications

#### Next Steps

- Implement tests for your current projects
- · Set up CI pipelines for automated testing
- · Practice mocking external dependencies
- · Focus on critical business logic coverage

#### Additional Resources

- pytest documentation
- unittest.mock guide
- · Jenkins pipeline tutorials
- Banking software testing standards

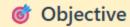
## **Questions & Discussion**

Let's discuss how these concepts apply to your current banking projects!

# Mini-Project: Personal Account Manager API

A Hands-On Training Session for First-Time Learners





Build and test a simple REST API for managing personal bank accounts using modern Python technologies. You'll create, read, update, and delete account information while learning industry best practices.



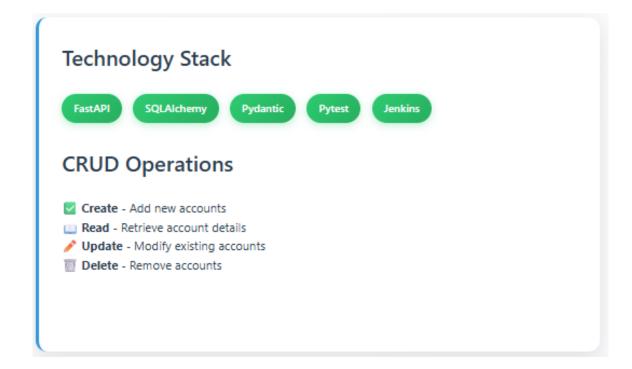
test\_with\_jenkinsfile.zip

#### What You'll Build

A **Personal Account Manager** - a simple REST API that allows users to perform basic CRUD operations on bank accounts.

## **Core Entity: Account**

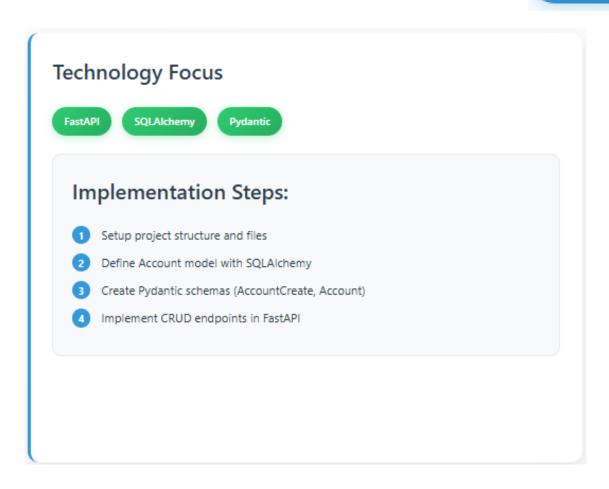
- id: Unique identifier (integer)
- **account\_holder:** Name of account holder (string)
- & balance: Current balance (float)
- account\_type: "Savings" or "Checking" (string)

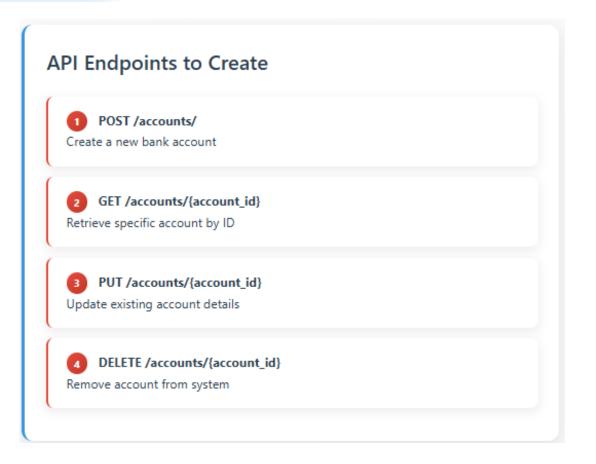




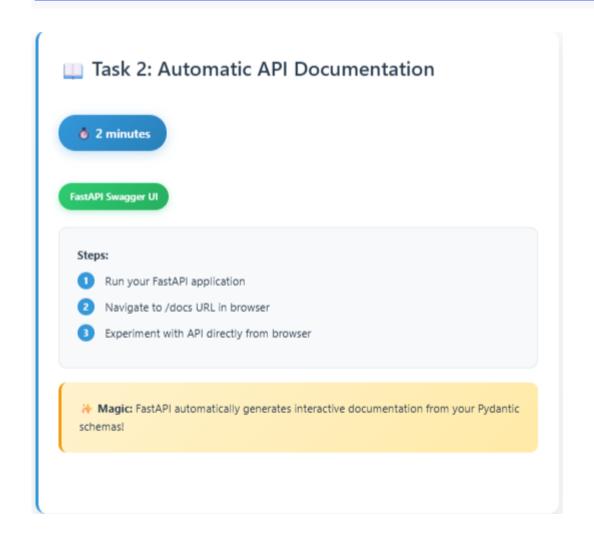
# **Task 1: The API Endpoints**

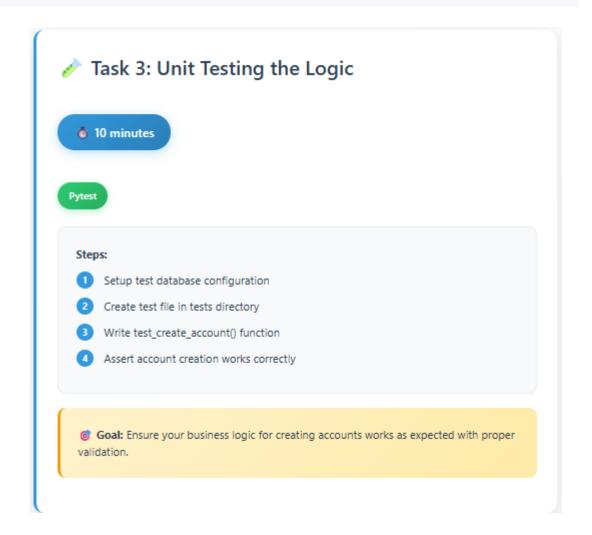
15-20 minutes













# Task 4: Automating with Jenkins



#### **Technology Focus**

Jenkinsfile

CI/CD Pipeline

#### Pipeline Stages to Create:

- Stage 1: Install Dependencies Run pip install -r requirements.txt
- Stage 2: Run Tests Execute tests using pytest command
- Stage 3: Build (Placeholder) Future build step (e.g., Docker image creation)

## **©** Key Concept: Pipeline as Code

The Jenkinsfile lives in your project repository and defines exactly how your code should be automatically tested and deployed. This means your build process is:

- Version Controlled Changes are tracked with your code
- Reusable Can be shared across teams
- Automated Runs automatically on code changes

What we accomplished: Built, documented, tested, and automated a fully functional microservice in under an hour!



#### Rapid Development with FastAPI

Used Python type hints to create robust APIs with minimal code. Fast to develop and fast to execute.



#### **Automatic Documentation**

Generated beautiful, interactive API documentation (Swagger UI) with zero extra effort - crucial for team collaboration.



#### Decoupled & Testable Logic

Separated database models, business logic, and API endpoints for cleaner, more testable code with Pytest.



#### Infrastructure as Code

Jenkinsfile provides repeatable, automated recipe ensuring applications are always tested and deployment-ready.



### Modern Python Stack Mastery

FastAPI + SQLAIchemy + Pydantic + Pytest forms a powerful, modern, and highly efficient stack for building scalable web applications and services.