Testing Suites Guide

Unit, Integration, and Regression Practices

1 Testing Pyramid & Purpose

A balanced automation suite treats unit, integration, and regression tests as collaborative layers instead of interchangeable files. Each layer answers a different question:

- **Unit tests** validate the smallest units (functions, methods, classes) in isolation so developers get fast feedback about business logic.
- **Integration tests** exercise how several components (or modules) work together, ensuring collaborations, protocols, and shared state behave as expected.
- **Regression tests** lock in fixes: every reported bug earns a guard so future work does not accidentally reintroduce the same failure.

Treating the pyramid as a whole keeps unit suites fast, integration suites lean, and regression suites focused on the rare but costly failures.

2 Unit Testing Best Practices

A good unit test suite is fast, deterministic, readable, and aligned with behaviour rather than implementation. Aim for single-purpose tests, clear names, and minimal reliance on environment or external services.

- **Control the scope:** each test should target one behaviour and avoid verifying multiple code paths at once.
- **Arrange–Act–Assert:** separate setup, execution, and assertions to keep tests easy to reason about.
- **Name tests explicitly:** e.g. `test_mean_raises_error_on_empty_input` makes intent obvious when a suite fails.
- **Share setup via fixtures or factories:** reuse helpers rather than copying boilerplate arrangement code.
- **Parameterize instead of duplicating:** reuse the same assertion logic across varied inputs.
- **Avoid hidden dependencies:** do not rely on global state, real databases, or the network; inject collaborators and stub external services.
- **Assert behaviour, not implementation:** focus on the public contract that should stay stable after refactors.

Arrange–Act–Assert in Practice

Listing 1: Clear AAA structure for a single behaviour

```python
def test_calculate_score_with_positive_values() -> None:
    # Arrange
    values = [10, 20, 30]
    expected = 20

    # Act
    result = calculate_score(values)

    # Assert
    assert result == expected
```

2.1 Python Example: Mean and Bank Utilities

These examples keep logic simple, rely on type hints, and use parameterization to cover important variants without extra test functions.

Listing 2: `math_utils.py` with explicit contracts

```python
from __future__ import annotations

from typing import Iterable


def mean(values: Iterable[float]) -> float:
    """Return the arithmetic mean of the provided sequence."""
    numbers = list(values)

    if not numbers:
        raise ValueError("mean() requires at least one value.")

    return sum(numbers) / len(numbers)
```

Listing 3: `tests/test_math_utils.py` demonstrating parameterized checks

```python
import pytest

from math_utils import mean


@pytest.mark.parametrize(
    "values,expected",
    [([1.0, 2.0, 3.0], 2.0), ([-1.0, 1.0], 0.0), ([5.5], 5.5)],
)
def test_mean_returns_expected_result(values: list[float], expected:
    float) -> None:
    result = mean(values)
    assert result == expected


def test_mean_raises_when_empty() -> None:
    with pytest.raises(ValueError):
        mean([])
```

Listing 4: `tests/test_bank.py` isolating account behaviours

```python
import pytest

from bank import BankAccount


@pytest.fixture
def base_account() -> BankAccount:
    return BankAccount(owner="Alex", balance=100.0)

```

```python

def test_deposit_increases_balance(base_account: BankAccount) -> None:
    base_account.deposit(50.0)
    assert base_account.balance == 150.0


def test_deposit_non_positive_amount_fails(base_account: BankAccount) -> None:
    with pytest.raises(ValueError):
        base_account.deposit(0)
```

3 JavaScript Example: Jest Modules and Clear Assertions

Jest supports the same best practices: favour descriptive `describe` blocks, `test.each` for repeated data, and `expect(...).toThrow` for error handling. Keep module exports tiny and explicit so tests import only what is needed.

Listing 5: `mathUtils.js` with explicit validation

```javascript
/**
 * Compute the arithmetic mean of an array of numbers.
 */
function mean(values) {
  if (!Array.isArray(values) || values.length === 0) {
    throw new Error("mean() requires a non-empty array of numbers.");
  }

  const total = values.reduce((acc, value) => {
    if (typeof value !== "number") {
      throw new Error("All elements must be numbers.");
    }
    return acc + value;
  }, 0);

  return total / values.length;
}

module.exports = { mean };
```

Listing 6: `mathUtils.test.js` using parameterized cases

```javascript
const { mean } = require("./mathUtils");

describe("mean", () => {
  test.each([
    [[1, 2, 3], 2],
    [[-1, 1], 0],
    [[5.5], 5.5],
  ])("returns %p for %p", (values, expected) => {
    expect(mean(values)).toBe(expected);
  });

  test("throws for empty array", () => {
    expect(() => mean([])).toThrow("mean() requires a non-empty array of numbers.");
  });
});
```

Listing 7: `bankAccount.test.js` isolating behaviours

```javascript
const { BankAccount } = require("./bankAccount");

describe("BankAccount", () => {
  test("deposit adds to balance", () => {
    const account = new BankAccount("Dana", 100);
    account.deposit(50);
    expect(account.balance).toBe(150);
  });

  test("deposit rejects non-positive amount", () => {
    const account = new BankAccount("Dana", 100);
    expect(() => account.deposit(0)).toThrow("Deposit amount must be a
       positive number.");
  });
});
```

4 Testing Playbook

A repeatable playbook turns the best practices above into reliable habits. Start with the unit layer for feedback and only climb the pyramid when you need to validate collaborations or guard a fixed bug.

Unit Testing Playbook

1. **Define the behaviour:** write down the contract, exceptions, and edge cases before touching code.
2. **Choose scope and fixtures:** limit each test to a single assertion and reuse factories or fixtures for shared setup.
3. **Follow AAA:** split Arrange, Act, and Assert so readers instantly understand the flow.
4. **Parameterize responsibly:** combine similar inputs into a single parametrized test instead of duplicating logic.
5. **Keep the surface stable:** assert the public API and avoid reaching into private helpers that may change during refactors.

Integration Testing Playbook

1. **Scope a scenario:** pick one workflow such as a transfer or API call that crosses module boundaries.
2. **Prepare real modules:** exercise the production code paths while stubbing only the truly external services.
3. **Control shared state:** reset databases, caches, or files between runs so tests stay deterministic.
4. **Assert contracts and side effects:** verify both the state changes and any notifications or outputs.

Regression Testing Playbook

1. **Reference the incident:** mention the ticket or issue ID so reviewers know why the test exists.
2. **Reproduce minimally:** encode only the precise steps and inputs required to trigger the bug.
3. **Lock randomness:** seed any stochastic helpers and avoid flaky external calls.

4. **Tag and guard:** keep regressions in dedicated folders or use explicit markers so teams can run them on demand.

Confidence Checklist

- Run fast unit suites on every commit so you catch regressions early.
- Reserve integration and regression suites for nightly jobs, feature branches, or explicit ‘pytest‘/‘npm‘ invocations.
- Keep an eye on the pyramid: the higher the layer, the fewer tests, but the broader the coverage.

5 Integration Testing Best Practices

Integration tests should exercise realistic interactions without becoming as slow or brittle as end-to-end suites. Keep them focused on meaningful scenarios, start and clean up state deliberately, and simulate only the external services that are too slow or unreliable for unit tests.

- **Scope a scenario:** test a single workflow, such as a transfer between accounts or a request hitting multiple modules.
- **Use real modules:** run the production code paths while faking only the truly external dependencies (APIs, queues).
- **Prepare and tear down consistently:** fixtures or factories should reset databases, files, or shared state between runs.
- **Label suites:** use `@pytest.mark.integration` or explicit Jest folders so you can run them separately when feedback speed matters.

5.1 Python Integration Example: Funds Transfer Flow

Listing 8: `transfer_service.py` coordinating withdrawals, deposits, and notifications

```python
from bank import BankAccount
from typing import Protocol


class Notifier(Protocol):
    def notify(self, from_owner: str, to_owner: str, amount: float) ->
       None:
        ...


def transfer_funds(
    source: BankAccount,
    destination: BankAccount,
    amount: float,
    notifier: Notifier,
) -> None:
    source.withdraw(amount)
    destination.deposit(amount)
    notifier.notify(source.owner, destination.owner, amount)
```

Listing 9: `tests/integration/test_transfer_flow.py` checking state and notification

```python
from bank import BankAccount
from transfer_service import transfer_funds


```

```python
class DummyNotifier:
    def __init__(self) -> None:
        self.messages: list[tuple[str, str, float]] = []

    def notify(self, from_owner: str, to_owner: str, amount: float) ->
       None:
        self.messages.append((from_owner, to_owner, amount))


def test_transfer_updates_balances_and_notifies() -> None:
    source = BankAccount(owner="Ali", balance=150.0)
    destination = BankAccount(owner="Bij", balance=50.0)
    notifier = DummyNotifier()

    transfer_funds(source, destination, 25.0, notifier)

    assert source.balance == 125.0
    assert destination.balance == 75.0
    assert notifier.messages == [("Ali", "Bij", 25.0)]
```

5.2 JavaScript Integration Example: Transfer Service

Listing 10: `transferService.js` wiring deposits, withdrawals, and notifications

```javascript
const { BankAccount } = require("./bankAccount");

function transferFunds(source, destination, amount, notifier) {
  source.withdraw(amount);
  destination.deposit(amount);
  notifier.notify(source.owner, destination.owner, amount);
}

module.exports = { transferFunds };
```

Listing 11: Jest integration test verifying balances and notification

```javascript
const { BankAccount } = require("./bankAccount");
const { transferFunds } = require("./transferService");

class SpyNotifier {
  constructor() {
    this.calls = [];
  }

  notify(fromOwner, toOwner, amount) {
    this.calls.push({ fromOwner, toOwner, amount });
  }
}

test("transfer updates balances and notifies owners", () => {
  const source = new BankAccount("Ali", 150);
  const destination = new BankAccount("Bij", 50);
  const notifier = new SpyNotifier();

  transferFunds(source, destination, 25, notifier);

  expect(source.balance).toBe(125);
  expect(destination.balance).toBe(75);
  expect(notifier.calls).toEqual([{ fromOwner: "Ali", toOwner: "Bij",
     amount: 25 }]);
});
```

6 Regression Testing Best Practices

Regression tests memorialize bugs by codifying the exact scenario that failed before. When a bug is fixed, add a regression test that reproduces the failure path, references the ticket, and stays narrow and deterministic.

- **Reference the bug:** include the ticket or issue number in the test name or a comment so reviewers understand the context.
- **Keep it minimal:** capture only the data and steps necessary to trigger the bug without extra noise.
- **Ensure repeatability:** remove randomness, seed fixtures, and avoid slow external calls.
- **Tag regressions:** use dedicated directories or markers (e.g. `tests/regression/` or `@pytest.mark.regression`) so CI can run them selectively.

6.1 Python Regression Example (Bug #1012)

Listing 12: `tests/regression/test_bug_1012.py` guarding against zero deposits

```python
import pytest

from bank import BankAccount


@pytest.mark.regression
def test_bug_1012_rejects_zero_deposit() -> None:
    account = BankAccount(owner="Eve", balance=100.0)

    with pytest.raises(ValueError):
        account.deposit(0.0)
```

6.2 JavaScript Regression Example (Issue JS-3281)

Listing 13: Regression test guarding overdrafts despite floating-point drift

```javascript
const { BankAccount } = require("./bankAccount");

test("JS-3281: withdraw fails when rounding noise would otherwise allow
    overdraft", () => {
  const account = new BankAccount("Frank", 100);
  account.withdraw(99.9999999999);
  expect(() => account.withdraw(0.0000000002)).toThrow("Insufficient
     funds.");
});
```

7 Organizing Tests and Execution

Keep your suites organised by purpose so you can run them independently.

Listing 14: Python project layout with dedicated suites

```
my_project/
  my_package/
    math_utils.py
    bank.py
  tests/
    unit/
      test_math_utils.py
      test_bank.py
    integration/
      test_transfer_flow.py
    regression/
      test_bug_1012.py
```

```
pytest tests/unit         # fast feedback while coding
pytest tests/integration # verify component collaborations
pytest tests/regression  # protect past bugs
```

For JavaScript projects, mirror the structure and leverage `package.json` scripts for each suite.

Listing 15: JavaScript project structure aligned with Jest suites

```
my-js-project/
  mathUtils.js
  bankAccount.js
  tests/
    unit/
      mathUtils.test.js
      bankAccount.test.js
    integration/
      transferService.test.js
    regression/
      bug_3281.test.js
  package.json
```

```
npm run test:unit
npm run test:integration
npm run test:regression
```

Tagging suites with markers or directories keeps CI pipelines lean, e.g. run only unit tests on every commit and reserve integration/regression suites for nightly pipelines or special branches.